UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Implementación de vehículo aéreo no tripulado y del software para su pilotaje y monitorización

Autor/es

ÓSCAR MARTÍNEZ MARTÍNEZ

Director/es

JESÚS MARÍA ARANSAY AZOFRA y JAVIER RICO AZAGRA ,

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2017-18

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Implementación de vehículo aéreo no tripulado y del software para su pilotaje y monitorización

Realizado por:

Óscar Martínez Martínez

Tutelado por:

Jesús María Aransay Azofra

Javier Rico Azagra

Logroño, junio, 2018

Resumen

El presente Trabajo Fin de Grado tiene el objetivo de construir nuestro propio dron, e implementar las bases de un software genérico de monitorización y pilotaje código abierto, que permita integrar y unificar distintos modos para pilotar.

Una vez construido nuestro dron, se ha desarrollado un software que vía wifi, controla al vehículo bien a través de un periférico de tipo teclado para PC o mediante gestos basándonos en la tecnología de "skeletal tracking" que nos ofrece un sensor Kinect® de Microsoft.

Como resultado final de este trabajo tenemos un software para escritorio que nos va a permitir monitorizar varios parámetros del vehículo, tales como su orientación en los tres grados de libertad, la calidad de la señal wifi que le está llegando, e incluso visualizar y grabar el vídeo que retransmite el dron con su cámara. Dicho software también será capaz de gestionar los distintos modos de pilotaje que hayan sido integrados. La integración de nuevos modos de pilotaje se vuelve más sencilla dado que hemos introducido un nivel de abstracción que permite transformar cualquier tipo de entrada a una orden de movimiento para el vehículo.

Palabras clave: Dron, software, pilotaje, monitorización, genérico, teclado, gestos.

Abstract

The aim of this Bachelor Thesis is to make our own drone and to implement the basis for a generic monitorization and flying open source software, which allows us to integrate and unify other future flying modes.

Once we had our drone built, we developed a software that manages the UAV[1] via wifi by using the two flight modes that we have also developed. It will be able to be flown using the peripheral keyboard of a PC or by body gestures using skeletal-tracking-based technologies that Microsoft Kinect® sensor provides.

As outcome, we have a desktop software that will allow us to monitor different parameters of the vehicle such as the orientation on its three degrees of freedom, the quality of the wifi signal, and also to see or record the video streaming from the drone. This software will permit us to manage the different flight modes that have been integrated too. The integration of new flying modes becomes easier since we have created an abstraction layer that transforms whatever input into a well-formed flight command for the drone.

Keywords: Dron, software, fly, monitor, generic, keyboard, gestures.

[1] Acrónimo anglosajón que refiere a un vehículo aéreo no tripulado.

Índice

Resumen 1
Abstract 1
Introducción 1
1. Antecedentes 1
2. Actores 1
2.1. Autoría 1
2.2. Tutores 2
3. Definición del sistema 2
3.1. Requisitos previos 2
3.2. Alcance 2
3.2.1. Alcance funcional 2
3.2.1.1. Vehículo 2
3.2.1.2. Software para monitorización 3
3.2.1.3. Software para pilotaje 3
3.2.1.4. Módulo de configuración 3
3.2.2. Alcance temporal 3
3.3. Dependencias 4
3.4. Exclusiones 4
4. Planificación 4
4.1. Definición, distribución y estimación temporal de tareas 4
4.2. Cronograma 6
4.3. Definición de metodologías 8
4.3.1. Metodología de gestión del proyecto 8
4.3.2. Metodología de gestión de versiones 9
4.4. Plan de contingencia 9
4.4.1. Identificación de riesgos 9
4.4.2. Identificación de soluciones 10
5. Análisis del sistema 11
5.1. Análisis de requisitos 11
5.2. Diagrama de Casos de Uso 11
5.3. Selección de tecnologías 17
5.3.1. Tecnologías para el desarrollo del software para monitorización 18
5.3.2. Tecnologías para el desarrollo del módulo de configuración 20
5.3.3. Tecnologías para el desarrollo del software para pilotaje 20
5.3.3.1. Pilotaje mediante teclado 21
5.3.3.2. Pilotaje mediante gestos 21
5.3.4. Tecnologías para el ensamblaje del vehículo 23
6. Implementación 25
6.1. Software para monitorización 25
6.1.1. Estación 26

6.1.1.1. Visualización de modelo 3D 26
6.1.1.2. Grabación y visualización de vídeo 31
6.1.1.3. Visualización de calidad de la señal 32
6.1.2. UAV 33
6.1.2.1. Envío de datos IMU 33
6.1.2.2. Envío de vídeo 34
6.1.2.3. Envío de información sobre la conexión 34
6.2. Software para pilotaje 35
6.2.1. Estación 36
6.2.1.1. Pilotaje mediante teclado 36
6.2.1.2. Pilotaje mediante gestos 37
6.2.2. UAV 44
6.3. Configuración y ensamblaje de UAV 45
7. Pruebas 47
7.1. Tests unitarios 47
7.1.1. Software para monitorización 47
7.1.2. Módulo de configuración 47
7.1.3. Software para pilotaje 47
7.2. Pruebas de integración 48
7.2.1. Software para monitorización 48
7.2.2. Software para pilotaje 50
8. Seguimiento y control 51
8.1. Duración real de las tareas 51
8.2. Cronograma real 53
8.3. Justificación de desviaciones 55
8.4. Riesgos materializados 56
8.5. Costes de material 56
9. Conclusiones 57
9.1. Objetivos alcanzados 57
9.2. Trabajo futuro 57
10. Bibliografía 59
11. Anexos 60

Introducción

El uso de drones como máquinas eficaces para desempeñar tareas que son muy costosas o imposibles de realizar desde tierra, se está extendiendo rápidamente en la actualidad abriéndose paso en los distintos campos e industrias (agrícola, audiovisual, seguridad, etc.). Estos vehículos parecen no tener limitaciones cuando hablamos de sus posibles aplicaciones.

Es por esto, que se hace cada vez más necesario el desarrollo de interfaces de usuario que hagan este producto más intuitivo y más fácil de manejar para todos.

Motivados por la necesidad de aprendizaje de las tecnologías aplicables a estos vehículos y porque creemos que puede ser un proyecto muy ameno, en este trabajo nos vamos a proponer desarrollar un sistema usable, que ofrezca la posibilidad de pilotar, configurar y monitorizar un UAV. Además, que sea escalable y permita incluir varios modos de pilotaje.

De cara al estudiante, debido a la envergadura de este trabajo, le va a permitir poner en práctica y ampliar la mayor parte de los conocimientos adquiridos en las asignaturas del plan de estudios, especialmente en el ámbito de la gestión de proyectos.

1. Antecedentes

A lo largo de mi estancia durante el periodo de prácticas extracurriculares en el Centro Tecnológico del Calzado de La Rioja (2017), participé en dos proyectos.

En uno de ellos hicimos uso de tecnologías de visión por computador para detección, reconocimiento, rastreo y medición de formas y objetos.

En el otro, desarrollamos una interfaz de usuario basada en la detección de gestos corporales, junto con una aplicación de realidad aumentada utilizando tecnologías para visión y creación de gráficos en dos y tres dimensiones.

Además, por cuenta propia he desarrollado diversas aplicaciones "*thin client*" para visión computerizada en sistemas embebidos.

2. Actores

2.1. Autoría

Este trabajo ha sido realizado por Óscar Martínez Martínez, estudiante de Grado en Ingeniería Informática en la Universidad de La Rioja.

2.2. Tutores

Este trabajo ha sido tutorizado por los profesores: Javier Rico Azagra del Departamento de Ingeniería Eléctrica y por Jesús María Aransay Azofra del Departamento de Matemáticas y Computación.

3. Definición del sistema

3.1. Requisitos previos

Para el desarrollo de este trabajo, es requisito imprescindible disponer de un UAV sobre el que implantar nuestro software. Para cumplir este requisito consideramos las siguientes dos opciones:

a) Elaborar nuestro propio vehículo.
b) Adquirir uno ya elaborado.

Con el fin de conseguir independencia de plataforma y desarrollo de software con propósitos más generales que los que ofrecería un sistema enfocado a una máquina específica, se ha decidido que el ensamblaje y configuración del vehículo será llevado a cabo como parte de este trabajo. De esta forma, podremos elegir y utilizar componentes y tecnologías extrapolables.

Puesto que los recursos con los que contamos son limitados, vamos a tratar de minimizar los costes al máximo a la hora de adquirir los componentes del vehículo. Para ello, trataremos de reutilizar materiales de los que disponemos de proyectos anteriores. Más adelante, en la sección 8.5, veremos los costes de todos los componentes. Esto tiene como finalidad servir de orientación para el lector que quiera construir su propio dron.

3.2. Alcance

3.2.1. Alcance funcional

3.2.1.1. Vehículo

El vehículo debe:

1. Ser capaz de mantenerse suspendido en el aire sin ayuda de ningún factor externo.
2. Tener una autonomía de vuelo de al menos 10 minutos.
3. Ser seguro, robusto y no poner en riesgo la integridad física de ningún actor o espectador.
4. Contar con:
    a. la configuración software adecuada para permitir el despliegue de los sistemas que se van a desarrollar.
    b. la programación de ejecución automática de programas necesarios para el funcionamiento de todo el conjunto del sistema.
5. Disponer de la configuración de red necesaria para garantizar la correcta y segura transferencia de información hacia la Estación.

6. Tener una cámara instalada y configurada para transmitir en vivo.

3.2.1.2. Software para monitorización

Este sistema comprende una arquitectura cliente-servidor que va a ser desplegada tanto en el vehículo, como en la Estación.

Ambas partes deben contar con la lógica necesaria para comunicarse y lograr:
1. Mostrar la orientación en tiempo real del vehículo con respecto a un punto fijo.
2. Mostrar la intensidad de la señal wifi.
3. Grabar y retransmitir en vivo el vídeo de la cámara que lleva acoplada el vehículo.

Con el fin de que el programa sea usable (en términos de rapidez y facilidad de interpretación), todos los datos que van a ser monitorizados han de presentarse de una manera clara y estructurada.

3.2.1.3. Software para pilotaje

Como en el anterior punto, este sistema también comprenderá dos partes:
1. Vehículo. El software desplegado en el vehículo debe:
    a. Recibir y ejecutar órdenes de movimiento.
    b. Ser tolerante a los siguientes fallos:
        i. Fallo de comunicación.
        ii. La Estación le comunique cualquier otro tipo de fallo (códigos de error[2]) y necesite abortar el vuelo.

2. Estación. El software alojado en la Estación debe:
    a. Permitir al usuario pilotar el vehículo mediante gestos, sin llevar acoplado al cuerpo ningún tipo de dispositivo.
    b. Permitir al usuario pilotar el vehículo mediante interacción con el teclado del PC.

3.2.1.4. Módulo de configuración

El sistema ha de incluir alguna manera de reconfigurar los parámetros necesarios para comunicarnos y monitorizar el vehículo desde la estación, así como variar la configuración básica de vuelo (modo de vuelo y acotar parámetros de vuelo).

Así mismo, dicho módulo deberá contar con algún tipo de persistencia, de manera que la configuración establecida sea recordada una vez que el sistema termine de ejecutarse.

3.2.2. Alcance temporal

El proyecto debe ser completado en una duración de alrededor de 300 horas, tal y como marca la guía docente de la asignatura.

[2] Son errores representados numéricamente que identifican un fallo en un sistema.

3.3. Dependencias

Para completar la fase de desarrollo y la fase de pruebas del software para monitorización, es indispensable disponer de los componentes hardware del vehículo (no hace falta que el vehículo esté armado en su totalidad).

Para completar las fases de desarrollo y pruebas del software de pilotaje necesitamos que el vehículo esté íntegramente armado y funcional.

Teniendo las dependencias identificadas, elaboraremos un plan de contingencia en el apartado 4.5, donde identificaremos las distintas situaciones que pueden poner en riesgo el cumplimiento de las dependencias de nuestro proyecto.

3.4. Exclusiones

1) Programación interna de chips de la circuitería del vehículo. Porque excederíamos las horas que se estiman en la guía docente.
2) Programación íntegra del módulo para comunicarnos vía MSP[3]. Porque sobrepasaríamos las horas estimadas y porque ya existe una versión de código abierto.
3) Documentación o manual de usuario sobre el manejo del sistema.

4. Planificación

4.1. Definición, distribución y estimación temporal de tareas

A continuación, veremos las tareas en las que dividiremos este proyecto, su fecha de comienzo y fin, y las horas totales que estimamos dedicar a cada una.

TAREA	COMIENZO	FIN	HORAS TOTALES
1. Definición del sistema	**5 de febrero**	**10 de febrero**	**13 horas**
1.1. Requisitos previos	5 de febrero	6 de febrero	1 horas
1.2. Alcance	6 de febrero	10 de febrero	10 horas
1.3. Dependencias	8 de febrero	9 de febrero	1 horas
1.4. Exclusiones	8 de febrero	9 de febrero	1 horas
2. Planificación	**12 de febrero**	**21 de febrero**	**20 horas**

[3] Multiwii Serial Protocol: protocolo mediante el cual nos comunicaremos con la controladora de vuelo.

2.1. Planificación temporal	14 de febrero	19 de febrero	10 horas
2.2. Definición de metodologías	19 de febrero	21 de febrero	8 horas
2.3. Plan de contigencia	21 de febrero	21 de febrero	2 horas
3. Análisis del sistema	**22 de febrero**	**29 de febrero**	**15 horas**
3.1. Análisis de requisitos	22 de febrero	24 de febrero	5 horas
3.2. Diagramas CdU	24 de febrero	27 de febrero	5 hora
3.3. Selección de metodologías	23 de febrero	26 de febrero	5 horas
4. Implementación	**1 de marzo**	**7 de mayo**	**159 horas**
4.1. Configuración y ensamblaje de UAV	1 de marzo	20 de abril	20 horas
4.2. Implementación SW monitorización	10 de marzo	5 de abril	50 horas
4.3. Implementación módulo de configuración.	26 de marzo	5 de abril	4 horas
4.4. Implementación SW pilotaje	5 de abril	7 de mayo	85 horas
5. Pruebas	**7 de mayo**	**15 de junio**	**40 horas**
5.1. Pruebas SW monitorización	7 de mayo	15 de mayo	5 horas
5.2. Pruebas SW pilotaje	15 de mayo	31 de mayo	35 horas
6. Seguimiento y control	**5 de febrero**	**21 de junio**	**43 horas**
6.1. Comparativa tiempos planificados y reales	5 de febrero	21 de junio	10 horas
6.2. Cronograma real	5 de febrero	21 de junio	10 horas
6.3. Justificación de desviaciones	25 de mayo	21 de junio	12 horas

6.4. Riesgos materializados	25 de mayo	21 de junio	1 horas
6.5. Reuniones	1 de marzo	21 de junio	10 horas
7.Memoria y anexos	**5 de febrero**	**21 de junio**	**40 horas**

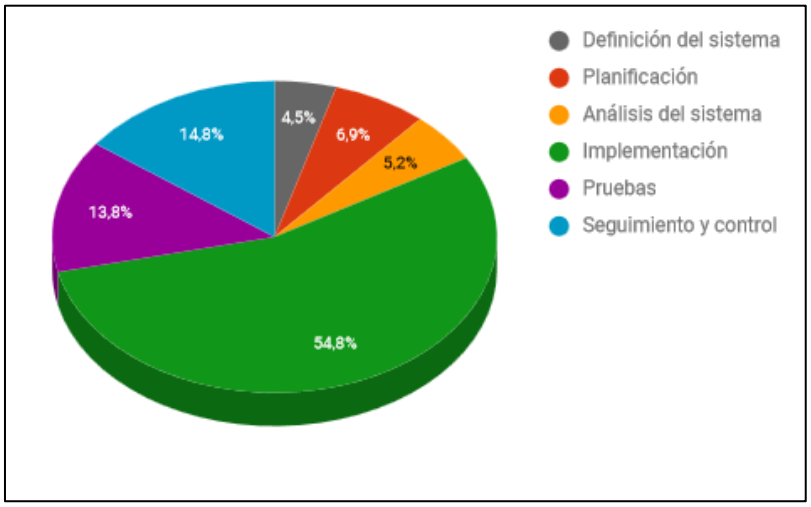

Figura 1. Proporción de dedicación estimada.

4.2. Cronograma

Seguidamente hemos elaborado un diagrama de Gantt para exponer el tiempo de dedicación previsto para las diferentes tareas a lo largo de los meses que va a durar este proyecto.

Tareas		Febrero																
		5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1.1	Requisitos previos																	
1.2.	Alcance																	
1.3.	Dependencias																	
1.4.	Exclusiones																	
2.1.	Planificación temporal																	
2.2.	Definición de metodologías																	
2.3.	Plan de contingencia																	
7.	Documentación y Anexos																	
6.3.	Justificación desviaciones																	

Tareas		Febrero							Marzo									
		22	23	24	25	26	27	28	1	2	3	4	5	6	7	8	9	10	11
3.1.	Análisis de requisitos																		
3.2.	Diagramas Casos de Uso																		
3.3.	Selección de metodologías																		
4.1.	Config. y ensamblaje de UAV																		
4.2.	Imp. Sw. Monitorización																		
7.	Documentación y Anexos																		
6.3	Justificación desviaciones																		

Tareas		Marzo																	
		12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
4.1.	Config. y ensamblaje de UAV																		
4.2.	Imp. Sw. Monitorización																		
4.3.	Imp. Módulo Configuración																		
7.	Documentación y Anexos																		
6.3	Justificación desviaciones																		

Tareas		Marzo		Abril																	
		30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
4.1.	Config. y ensamblaje de UAV																					
4.2.	Imp. Sw. Monitorización																					
4.4.	Imp. Sw. Pilotaje																					
4.3.	Imp. Módulo Configuración																					
7.	Documentación y Anexos																					
6.3.	Justificación desviaciones																					

Tareas		Abril										Mayo										
		20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7	8	9	10	11
4.4.	Imp. Sw. Pilotaje																						
5.1.	Pruebas Sw. Monitor.																						
7.	Documentación y Anexos																						
6.3	Justificación desviaciones																						

Tareas		Mayo																				
		12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
5.1.	Pruebas Sw. Monitor.																				
5.2.	Pruebas Sw. Pilotaje																				
7.	Documentación y Anexos																				
6.3	Justificación desviaciones																				

Tareas		Junio																				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
7.	Documentación y Anexos																					
6.3.	Justificación desviaciones																					

4.3. Definición de metodologías

4.3.1. Metodología de gestión del proyecto

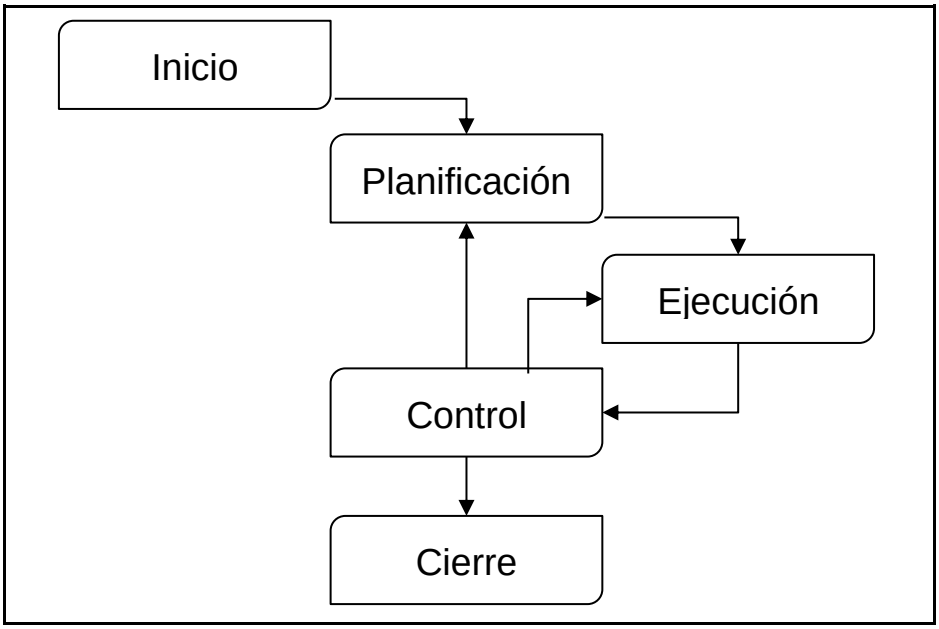

Figura 2. Fases del proyecto.

La metodología elegida es una metodología tradicional. A lo largo de la asignatura de Proyectos de Informática, se impartió dicha metodología siguiendo la guía *PMBoK* publicada por *Project Management Institute* (*PMI*). Ésta sigue el flujo de trabajo descrito en la figura 2:

1. Inicio.
2. Planificación: Definición de alcance y recursos.
3. Ejecución: Desarrollo del software.
4. Seguimiento y control: Documentar el proceso de desarrollo, verificar que se cumple la planificación y plasmar las desviaciones o cambios introducidos con respecto a la planificación.
5. Cierre: Finalización de todos los flujos anteriores y entrega del proyecto.

Estamos ante un proyecto de carácter individual, es decir, no implica ninguna forma de comunicación ni trabajo en equipo. Además, puesto que el cliente de este proyecto es el que escribe, no tendremos que lidiar con este aspecto. Por estas razones, decidimos decantarnos por la metodología aprendida en la asignatura antes nombrada.

4.3.2. Metodología de gestión de versiones

En cuanto a la gestión de las versiones del código, utilizaremos Git. Gestionaremos todo el código y sus cambios de manera remota, para ello, utilizaremos el servicio en la nube gratuito de GitHub.

Las razones por las que hemos tomado esta decisión son las siguientes:

- Facilitar la evaluación y el seguimiento del proyecto por parte de los tutores y del tribunal.
- Es gratuito.
- Todo el software desarrollado será publicado en GitHub de modo que podrá ser públicamente accesible y con el fin de contribuir al desarrollo de futuros proyectos.
- Facilitar la movilidad. Puesto que no necesitamos llevar un soporte físico con el código, simplemente descargaremos la versión necesaria de la nube.
- Tenemos experiencia gestionando las versiones de anteriores proyectos con esta plataforma.

4.4. Plan de contingencia

En este apartado analizaremos las tareas que son susceptibles de algún imprevisto. Identificaremos sus riesgos y propondremos soluciones para conseguir minimizar su impacto y garantizar la continuidad del proyecto.

4.4.1. Identificación de riesgos

En la siguiente tabla veremos la identificación de las posibles situaciones de riesgo a las que nos podemos enfrentar durante el desarrollo de este trabajo y el impacto que se prevé que tengan en caso de llegar a materializarse.

Fuente	Riesgo	Impacto
Implementación UAV	Componente hardware defectuoso	Medio
	Componente hardware dañado	Medio
	Componente del vehículo no adecuado	Alto
Equipo de trabajo	Daños irreparables en equipo de trabajo	Alto
	Pérdida o robo del equipo de trabajo	Alto
Desarrollo software	Pérdida de datos del disco duro	Medio

	Robo o pérdida de cuenta de GitHub	Alto

4.4.2. Identificación de soluciones

Fuente	Riesgo	Si sucede	Para evitar/minimizar
Implementación UAV	Componente HW defectuoso	Adquirir un repuesto	Disponer de repuestos suficientes. Al menos, un duplicado.
	Componente HW dañado		
	Componente no adecuado	Concluir la razón por la que no es adecuado y adquirir uno que sí cumpla.	Estudiar previa y exhaustivamente qué componentes pueden dar el resultado óptimo.
Equipo de trabajo	Pérdida o robo	Adquirir otro equipo tan pronto como sea posible para continuar el proyecto	Guardar el equipo en un lugar seguro cuando haya que desplazarse
	Daños irreparables		Usar funda robusta que prevenga golpes fuertes. No prestar el equipo a nadie sin nuestra supervisión.
Desarrollo software	Pérdida de datos del disco duro	Descargar última versión subida a la nube.	Evitar infección malware mediante antivirus y cortafuegos. Utilizar almacenamiento en la nube.
	Robo o eliminación de cuenta de GitHub	Obtener última versión guardada en un dispositivo físico	Utilizar almacenamiento físico.

5. Análisis del sistema

5.1. Análisis de requisitos

A continuación, se muestran las necesidades del sistema software. Para facilitar el proceso, vamos a utilizar las técnicas de análisis y diseño de software aprendidas en la asignatura de Diseño Tecnológico de Sistemas de Información que se basan en el modelado UML.

5.2. Diagrama de Casos de Uso

A continuación, se presenta el análisis de comportamiento del sistema identificando actores y acciones que llevarán a cabo.

En primer lugar, vamos a identificar los actores. Nuestro sistema distribuido consiste en la comunicación bidireccional entre una Estación y un *UAV*, así que, claramente estos dos serán actores del sistema. Además, puesto que nuestra Estación no es autónoma, necesitará que una persona se encargue de gestionar, tanto el pilotaje (piloto), como la configuración de la aplicación (usuario).

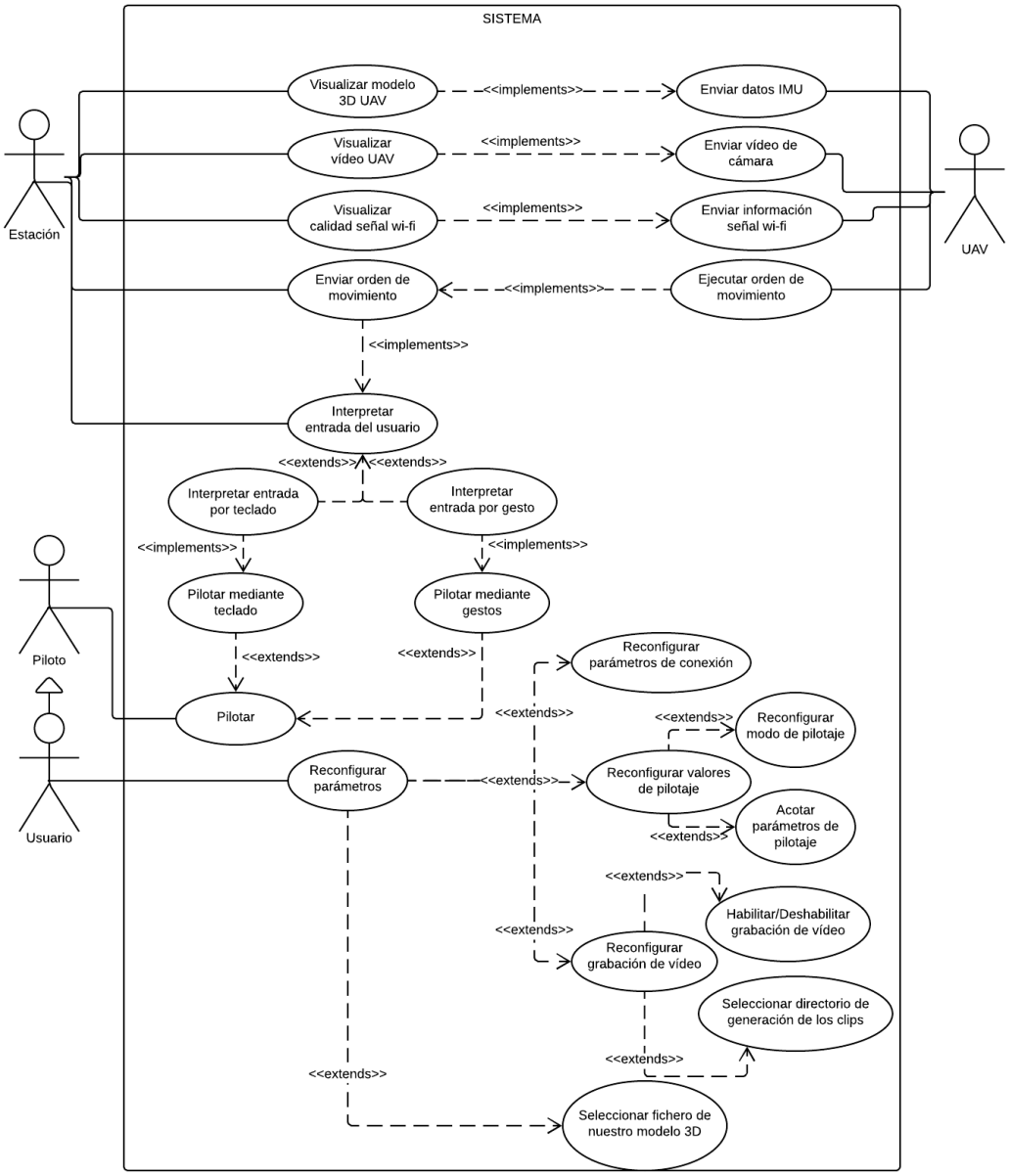

Figura 3. Diagrama de casos de uso.

Con la finalidad de obtener mayor detalle, a continuación, especificaremos brevemente cada caso de uso.

Caso de uso	Visualizar modelo
Actores	Estación
Resumen	Estación muestra gráficamente un modelo 3D de un UAV que siga la orientación del vehículo en tiempo real.
Precondición	• Conexión establecida y controladora de vuelo conectada. • Modelo 3D cargado en memoria y listo para visualizarse.
Postcondición	• Modelo 3D cambia su orientación de acuerdo al modelo real.
Dependencia	Enviar datos IMU

Caso de uso	Enviar datos IMU[4]
Actores	UAV
Resumen	Computadora embebida en el vehículo recoge los datos suministrados por los sensores de la placa y los envía vía wifi, a la computadora de la Estación.
Precondición	• Conexión establecida. • Controladora de vuelo conectada. • Giróscopo y acelerómetro activos.

Caso de uso	Visualizar vídeo UAV
Actores	Estación
Resumen	Estación visualiza por pantalla el vídeo que restransmite el vehículo en tiempo real.
Precondición	Conexión establecida.
Postcondición	Se muestra el vídeo en tiempo real por pantalla.
Dependencia	Enviar vídeo cámara

[4] Información inercial del vehículo suministrada por el acelerómetro y giróscopo.

Caso de uso	Enviar vídeo de cámara
Actores	UAV
Resumen	El computador embebido obtiene los frames[5] de la cámara que tiene instalada y los envía vía wifi a la computadora de la Estación.
Precondición	• Conexión establecida. • Cámara instalada y preparada para capturar.
Postcondición	

Caso de uso	Visualizar calidad wifi
Actores	Estación
Resumen	Estación grafica en tiempo real la intensidad de la señal wifi que le llega la placa de red del vehículo.
Precondición	• Conexión establecida. • Gráfico preparado para graficar.
Postcondición	Se grafica el valor de la intensidad de la señal wifi en ese momento.
Dependencia	Enviar información señal wifi

Caso de uso	Enviar información señal wifi
Actores	UAV
Resumen	Computadora embebida en vehículo transmite la información necesaria sobre la red wifi a la computadora de la Estación vía wifi.
Precondición	Conexión establecida.

Caso de uso	Enviar orden de movimiento
Actores	Estación
Resumen	Estación envía un paquete que contiene la orden que debe ejecutar el vehículo vía wifi.
Precondición	Conexión establecida.

[5] Cuadro de pixeles o imagen.

Caso de uso	Ejecutar orden de movimiento
Actores	UAV
Resumen	La computadora embebida del vehículo envía la orden a la controladora para que la ejecute.
Precondición	• Conexión establecida. • Controladora de vuelo conectada. • Motores conectados.
Postcondición	El vehículo se mueve conforme el piloto le ha indicado.
Dependencia	Enviar orden de movimiento

Caso de uso	Interpretar entrada por teclado
Actores	Estación
Resumen	Estación interpreta la entrada que el piloto le pasado a través del teclado y elabora una orden interpretable por el vehículo.
Precondición	Suministro de valores por teclado.
Postcondición	Elaboración de orden de movimiento para el vehículo.
Dependencia	Pilotar mediante teclado

Caso de uso	Interpretar entrada por gesto
Actores	Estación
Resumen	Estación interpreta las posiciones de las articulaciones del piloto y la mapea a una orden que el vehículo pueda entender.
Precondición	• Usuario realiza un gesto. • Posiciones de articulaciones captadas.
Postcondición	Elaboración de orden de movimiento para el vehículo.
Dependencia	Pilotar mediante gestos

Caso de uso	Pilotar mediante teclado
Actores	Piloto
Resumen	El usuario a través de una interfaz gráfica y periférico de tipo teclado puede modificar en tiempo real los parámetros que quiere suministrar al vehículo para que ejecute un movimiento.
Precondición	Conexión establecida
Postcondición	Valores de movimiento actualizados.

Caso de uso	Pilotar mediante gestos
Actores	Piloto
Resumen	El piloto, posicionado frente a un sensor, realiza un set de gestos previamente establecidos para manejar el vehículo.
Precondición	• Conexión establecida • Sensor activado • Piloto ha realizado un gesto

Caso de uso	Seleccionar fichero de nuestro modelo 3D
Actores	Usuario
Resumen	El usuario busca y selecciona la ubicación de un fichero de un modelo 3D.
Precondición	La ruta del fichero existe.
Postcondición	La ruta del fichero es guardada para luego poder ser cargado a memoria.

Caso de uso	Seleccionar directorio de generación de los clips
Actores	Usuario
Resumen	El usuario busca y selecciona un directorio en el disco donde se van a generar los clips de vídeo.
Precondición	La ruta del directorio existe.
Postcondición	La ruta del directorio es guardada para luego poder ser usada.

Caso de uso	Habilitar/Deshabilitar grabación de vídeo
Actores	Usuario
Resumen	El usuario habilita o deshabilita la grabación de vídeo en tiempo de vuelo.
Precondición	Debe haber guardada una ruta de generación de clips.
Postcondición	Se guarda la opción elegida.

Caso de uso	Acotar parámetros de pilotaje
Actores	Usuario
Resumen	El usuario modifica el valor mínimo, máximo y medio de los parámetros de vuelo: Yaw, Pitch, Roll, Throttle.
Precondición	Debe haber guardada una ruta de generación de clips.
Postcondición	Se guarda la opción elegida.

Caso de uso	Reconfigurar modo de pilotaje
Actores	Usuario
Resumen	El usuario elige el modo de pilotaje que quiere utilizar.
Precondición	
Postcondición	Se cambia el modo de pilotaje de la aplicación.

Caso de uso	Reconfigurar parámetros de conexión
Actores	Usuario
Resumen	El usuario definela IP del vehículo y los puertos necesarios para la comunicación.
Precondición	
Postcondición	Se guardan dichos parámetros.

5.3. Selección de tecnologías

A continuación, basándonos en los requerimientos del sistema, vamos a realizar un estudio y una selección de las mejores tecnologías que podemos utilizar para llevar a cabo el sistema previamente descrito.

Puesto que no tenemos un cliente que nos restrinja el uso de determinadas tecnologías, vamos a elegir las que creemos más adecuadas, manteniendo siempre la premisa de elaborar un software libre y basándonos en la experiencia de anteriores proyectos de carácter similar.

5.3.1. Tecnologías para el desarrollo del software para monitorización

Comenzamos por identificar los parámetros que queremos monitorizar:

- Orientación del vehículo.
- Vídeo de la cámara del vehículo.
- Calidad de la señal wifi que llega al vehículo.

Para poder monitorizar los anteriores parámetros (comunicarlos entre el UAV y la Estación), debemos seleccionar:

- Protocolo de comunicación
- Lenguaje de programación.
- API para visualizar gráficos 3D.
- API para tratamiento de imágenes.

Protocolo de comunicación.

Vamos a seleccionar el protocolo mediante el cual viajará nuestra información. Para enviar la información vamos a utilizar sockets. Como hemos visto en la asignatura de la titulación denominada Sistemas Distribuidos, la programación en red con sockets es muy útil cuando queremos implementar protocolos de comunicación propios y lograr buenos tiempos de respuesta.

La comunicación va a ser unidireccional porque va a ser el vehículo, el nodo activo que va a estar continuamente enviando paquetes sobre el estado en el que se encuentra.

Tenemos dos alternativas en cuanto a tipo de sockets para elegir.

A. UDP (Protocolo Universal de Datos).
B. TCP (Protocolo de Control de Transmisión).

Entre estos dos protocolos queremos elegir el más rápido, no nos importa especialmente la seguridad de que los paquetes lleguen en orden o sin errores, puesto que simplemente estamos desarrollando una herramienta de monitorización en tiempo real. Si por ejemplo un frame de vídeo se pierde, no sería un problema puesto que estaremos recibiendo alrededor de 40 frames por segundo. Por esta razón elegiremos UDP como protocolo de comunicación.

Lenguaje de programación.

En cuanto a lenguajes de programación, debemos elegir el que usaremos en la Estación y el que usaremos en la computadora del vehículo. Ambos lenguajes deben tener una API de sockets que incorpore soporte para UDP.

Para el computador del vehículo necesitamos un lenguaje sencillo pero que a su vez incorpore muchas bibliotecas o extensiones puesto que vamos a necesitar acceder a la cámara, a la controladora, etc. y posiblemente a otros periféricos que añadamos en un futuro (sensores, servos, etc.). Uno de los lenguajes que nos pueden facilitar mucho la tarea de desarrollo en este tipo de entorno es Python. Posiblemente C++ puede ofrecernos mejor optimización de recursos, sin embargo, puesto que nuestro sistema está pensado para que la mayor parte de la carga de procesamiento se delegue en la Estación, nos decantaremos por Python y su simplicidad.

Para la computadora de la Estación, necesitaremos un lenguaje potente, con API de sockets UDP y posibilidad de desarrollar una interfaz gráfica que soporte gráficos 3D, imágenes y gráficos estadísticos. Una buena opción sería C# o Java. Puesto que el estudiante ha desarrollado proyectos con gráficos 3D y vídeo en C#, elegiremos esta opción.

API para visualizar gráficos 3D.

Respecto al visualizado de gráficos 3D, existen dos alternativas principales:

- DirectX[6].
- OpenGL[7].

Ambas son compatibles con C# (OpenGL necesitaría un *wrapper*[8], puesto que no existe una API nativa). Es cierto que OpenGL está actualmente más optimizado que DirectX, además OpenGL es multiplataforma y, sin embargo, DirectX solo puede correr bajo una máquina Windows. Por los aspectos nombrados anteriormente, junto con que hemos trabajado anteriormente con gráficos en OpenGL, nos decidiremos por esta última.

Podemos ver una comparativa más extensa entre estas dos APIs en el siguiente enlace:

https://es.wikipedia.org/wiki/Comparaci%C3%B3n_entre_Direct3D_y_OpenGL

OpenGL es una API para la producción de gráficos por ordenador, una característica que la hace interesante para nuestro proyecto, es que, a pesar de estar desarrollada íntegramente en C, existen adaptaciones (*wrappers*) para casi todos los lenguajes más usados.

API para tratamiento de imágenes.

Veremos ahora qué API elegimos para el tratamiento de imágenes. La idea de este proyecto es servir de base para futuros proyectos de visión computerizada en vehículos aéreos no tripulados. Por ello, será una buena decisión utilizar alguna librería de visión por computador desde un principio para visualizar las imágenes que manda el vehículo a la Estación. Una biblioteca muy utilizada y con muy buenos resultados es OpenCV. Al igual que OpenGL, está

[6] https://en.wikipedia.org/wiki/DirectX
[7] https://www.khronos.org/opengl/wiki/
[8] En términos de software, es un envoltorio que se realiza a una biblioteca de funciones para poder ser utilizada en otro lenguaje para el cual no fue desarrollado. Esto implica un cambio de tipos y en ocasiones, una gestión más explícita de memoria.

desarrollada en C/C++ pero existen *wrappers* para casi todos los lenguajes. Por ello decidimos utilizar OpenCV tanto en el vehículo para obtener las imágenes de la cámara, como en la Estación para visualizarlas. En el caso de la Estación tendremos que utilizar un wrapper para C# denominado EmguCV[9], muy optimizado y que ofrece la posibilidad de invocar funciones nativas de la biblioteca genuina.

5.3.2. Tecnologías para el desarrollo del módulo de configuración

Como lenguaje de programación elegiremos C#, puesto que este módulo va a ir acoplado al software de monitorización. La idea inicial es que este módulo esté integrado a modo de menú dentro de la misma aplicación. Dicho módulo nos permitirá guardar los parámetros de configuración para monitorizar y pilotar el vehículo.

A la hora de decidir la tecnología para implementar la persistencia podemos decantarnos por muchas opciones. Entre ellas las más adecuadas:

A. Bases de datos.
B. XML en ficheros.
C. JSON en ficheros.
D. Texto plano en ficheros.

Vamos a utilizar dos criterios para seleccionar la más adecuada:

- Cantidad de datos a guardar.
- Claridad o trazabilidad del fichero generado.

En lo que se refiere a la cantidad de datos a guardar, debemos fijarnos en el apartado de análisis de requisitos. Más específicamente en el diagrama de casos de uso. Vemos que la cantidad de datos que tenemos que almacenar no es de gran dimensión.

El usuario final no va a utilizar el fichero de configuración para nada ya que la configuración tiene que ser automáticamente leída y escrita por la aplicación. La trazabilidad que buscamos es únicamente útil para el desarrollador durante la fase de pruebas.

Puesto que en la asignatura de Sistemas Distribuidos hemos aprendido a leer y escribir ficheros XML y este caso de uso puede ser un escenario adecuado para ponerlo a prueba, nos decantamos por esta tecnología. En esta asignatura hemos trabajado ficheros XML mediante árboles DOM en Java. Utilizaremos una tecnología homóloga para .NET.

5.3.3. Tecnologías para el desarrollo del software para pilotaje

El desarrollo del módulo de la Estación tendremos que abordarlo pensando en las dos funcionalidades que queremos:

A. Pilotaje mediante teclado del PC.
B. Pilotaje mediante gestos.

[9] http://www.emgu.com/wiki/index.php/Main_Page

Cada uno de estos tipos de pilotaje darán lugar a una biblioteca de funciones distinta, por lo que las tecnologías no necesariamente han de ser las mismas.

Sin embargo, hay una parte que va a ser común a ambas implementaciones, el protocolo de comunicación. Por ello, al igual que en el apartado anterior sobre el software de monitorización, debemos decidir qué protocolo vamos a utilizar. De nuevo tenemos dos alternativas principales:

C. UDP (Protocolo Universal de Datos).
D. TCP (Protocolo de Control de Transmisión).

Vamos a elaborar un estudio teniendo en cuenta nuestras necesidades y viendo qué protocolo se adapta mejor.

La primera característica que analizaremos está relacionada con la detección de errores en los paquetes y la corrección de los mismos. Esta característica es importante porque es necesario que todas las órdenes de movimiento lleguen perfectamente al vehículo para no provocar ningún movimiento no controlado. Tanto TCP como UDP detectan paquetes erróneos, sin embargo, solo TCP los corrige.

Otro factor a tener en cuenta es la velocidad de la comunicación. Aunque UDP destaca por ser más rápido que TCP, debemos tener en cuenta que los paquetes son muy poco pesados (menos de 32 bytes), puesto que simplemente están formados por cuatro números enteros. Así que usar uno u otro no tendrá un impacto notable en el sistema. Como veremos más adelante, los valores del paquete del que hablamos representarán la orden de movimiento que se le pasará al UAV.

Finalmente optamos por TCP, por la seguridad que nos ofrece durante la comunicación.

5.3.3.1. Pilotaje mediante teclado

La decisión que tenemos que tomar ahora corresponde con el lenguaje de programación. Necesitamos uno que nos permita acceder mediante algunas funciones al estado de las teclas del periférico (si están pulsadas o no). El lenguaje que elegimos es C, porque nos puede proporcionar tiempos de respuesta muy bajos y sencillas funciones del sistema que monitorizan el estado de las teclas.

5.3.3.2. Pilotaje mediante gestos

Como anteriormente hemos puntualizado, el pilotaje mediante gestos debe realizarse de manera que el usuario no tenga que portar ningún tipo de dispositivo. Esta necesidad implica que el seguimiento y procesamiento de los movimientos del usuario se realicen mediante algún tipo de sensor óptico.

Las decisiones que deberemos tomar para desarrollar este módulo de pilotaje van a girar en torno a los siguientes aspectos:

- Sensor.
- SDK para gestionar el sensor.
- Lenguaje de programación.

Existen numerosos fabricantes y marcas de sensores que cuentan con la tecnología de "*skeletal tracking*". Denominamos "*skeletal tracking*" al conjunto de algoritmos de visión por computador que, combinados con métodos de inteligencia artificial, consiguen detectar cuerpos humanos y estimar de manera precisa la posición espacial de algunas de sus articulaciones.

Algunos de los sensores que nos pueden servir para lograr nuestro objetivo pueden ser:

- VicoVR.
- Orbbec.
- Kinect.

Los tres nos valen, puesto que todos incorporan la tecnología que necesitamos. Pero veamos más a fondo alguna característica más para tomar una decisión.

VicoVR es más indicado para aplicaciones móviles. Incorpora mucha documentación y una completa SDK para sistemas operativos móviles como pueden ser Android o iOS.

Siendo Kinect y Orbbec los más indicados para nuestra plataforma, nos decantamos por Kinect de Microsoft por el coste económico. Podemos adquirir un sensor Kinect por alrededor de 45 euros, frente al sensor Orbbec que se puede encontrar en el mercado por la cuantía de 150 euros.

Veamos ahora qué SDK seleccionamos para gestionar el sensor de Kinect. En este aspecto sí que conocemos a fondo las dos principales alternativas, pues el estudiante ha estado desarrollando un probador virtual de zapatos, durante el periodo de prácticas en empresa, usando estas tecnologías.

Las alternativas de las que hablamos son las siguientes:

- Kinect SDK[10].
- OpenNI SDK[11].

Ambas nos ofrecen una API muy completa y con ambas podríamos lograr nuestro objetivo. Realizaremos una tabla comparativa de las características más importantes (aprendido en la asignatura de Taller Transversal I).

Características	**Kinect SDK**	**OpenNI SDK**
Gratuita	Sí.	Sí.
Código abierto	No.	Sí.
Lenguajes compatibles	C++, C#, Visual Basic.	C, C++, C#, Python, Java
Sistemas operativos compatibles	Microsoft Windows.	Multiplataforma.

[10] https://docs.microsoft.com/en-us/previous-versions/windows/kinect
[11] http://openni.ru/openni-programmers-guide/index.html

Documentación y soporte	Documentación decente y bastantes ejemplos de código.	Mucha documentación. Comunidad grande de usuarios que comparten sus proyectos y experiencias.

Vemos que, aunque ambos proporcionan la misma funcionalidad, en términos no funcionales difieren bastante, quedando reflejado que OpenNI es bastante más completa y escalable.

Además, es importante notar un último hecho que no ha quedado reflejado en la tabla. Si utilizáramos Kinect SDK, nos vemos obligados a usar para siempre un sensor Kinect. Por el contrario, si usamos OpenNI, podremos cambiar de sensor y nuestro código seguirá siendo válido, puesto que incorpora un nivel de abstracción más, con el fin de conseguir mayor compatibilidad con los distintos sensores que existen en el mercado.

Entonces, nos decantamos por OpenNI SDK. Ahora debemos elegir el lenguaje de programación en el que queramos implementar nuestro módulo. Es obvio que debemos elegir entre los lenguajes compatibles que se han mostrado en la tabla:

- C/C++.
- C#.
- Python.
- Java.

Todos ellos disponen de API de sockets TCP, así que todos son válidos. Y todos pueden utilizar la SDK de OpenNI. Así que vamos a buscar el lenguaje que nos dé mayor eficiencia computacional, porque, aunque hoy en día los lenguajes han evolucionado mucho y las diferencias entre ellos son muy pequeñas, en términos de visión computerizada sí que podemos encontrar grandes diferencias.

Puesto que OpenNI SDK está desarrollada íntegramente en C++, decidimos utilizar C++ para desarrollar nuestro programa. Esto implica que no tengamos que usar un *wrapper* para otro lenguaje, que, por lo general, suelen tener un rendimiento más bajo que la API original.

En cuanto a la parte que va a ir desplegada en el vehículo hay pocas decisiones que tomar. Solamente necesitamos un lenguaje de programación que sea capaz de recibir los paquetes de las órdenes y ejecutarlos sobre la placa. Por ello elegiremos Python, por la facilidad de integrar mucha utilidad con pocas líneas de código y porque es perfecto para desplegar sobre un computador embebido de estas características.

5.3.4. Tecnologías para el ensamblaje del vehículo

En este apartado veremos los componentes electrónicos que utilizaremos para montar el UAV. En concreto, debemos decidir sobre los siguientes aspectos:

1. Número de ejes.
2. Chasis.
3. Batería.
4. Motores.

5. Variadores.
6. Controladora de vuelo.
7. Computador a bordo.
8. Placa distribuidora de voltaje.
9. Hélices.

En cuanto al número de ejes, lo más común es un cuadricóptero. Da la estabilidad suficiente y hay que tener en cuenta que cuantos más ejes, el precio del material aumenta. Por ello, nos decantamos por construir un vehículo de cuatro ejes.

Las decisiones que se toman a continuación se basan en las recomendaciones que ofrece el siguiente artículo publicado por la web de noticias y recomendaciones sobre vehículos de radiocontrol RC FPV Plane:

http://rcfpvplane.com/quadcopter-frame-sizes-guide/

El chasis que vayamos a adquirir debe ser de cuadricóptero. Lo que debemos decidir es el tamaño del chasis, ya que esto nos va a condicionar las características de los componentes que decidiremos más adelante. En principio, no tenemos limitación en cuanto a dimensiones. Pero iremos a por uno cuyas dimensiones sean estándar con el fin de encontrar repuestos fácilmente en caso de sufrir daños. Así que elegimos un chasis de dimensión 250 milímetros (medidos de eje a eje).

La batería que utilizaremos tiene que ir acorde con el tamaño del vehículo. Al ser un vehículo de tamaño medio-pequeño, no podemos excedernos con el peso de la batería. Además, tenemos que tener en cuenta el requisito 2 del apartado de alcance funcional del vehículo (*el vehículo debe tener una autonomía de vuelo de al menos 10 minutos*). Por estas dos razones nos decantamos por una batería de polímero de litio con 2200 mAh y 3 celdas.

Los motores tienen que ser los indicados para esta batería, y que den el par suficiente como para elevar fácilmente el vehículo. Con unos motores de 2300 kv será suficiente.

Los variadores son los componentes que se encargan de administrar la energía necesaria a los motores según le va informando la placa controladora. Son un componente esencial y debemos asegurarnos de que combinan perfectamente con nuestros motores y con la batería. Elegiremos unos BL-Heli de 30 A.

La controladora de vuelo será la que se encargue de sostener el vehículo en el aire. Teniendo en cuenta la información que recibe de sus propios sensores es capaz de enviar las órdenes a los motores para que el vehículo se mantenga plano en el aire. En nuestro caso, ya contamos con una controladora Naze32, utilizaremos esa misma para no tener que adquirir una nueva.

El computador a bordo necesita módulo wifi y puerto USB para alimentar y comunicarse con la controladora. Contamos con una Raspberry Pi 3 que ya hemos utilizado en otros proyectos de robótica. Como cumple con lo que necesitamos, utilizaremos ésta.

La placa distribuidora de voltaje es un circuito que se encarga de distribuir la energía de la batería a los distintos componentes conectados. En nuestro caso necesitaremos conectar los cuatro variadores y la Raspberry Pi. Por ello es importante que tenga cuatro salidas para los variadores sin limitación de voltaje y al menos una salida de 5 voltios y 3 amperios para la Raspberry Pi. Elegiremos una Maltek Minipower Hub.

Por último, tendremos que elegir unas hélices adecuadas al tamaño de nuestro chasis para que no dañen nuestros componentes a la hora de girar y sean capaces de ofrecer el empuje necesario para elevar el UAV. Además, deben ir acorde con los motores para que éstos no sufran. Basándonos en el siguiente artículo:

http://blog.rc-fever.com/2012/11/relationship-between-motor-propeller-esc-and-battery

Decidimos utilizar unas hélices bipala con 35 grados de inclinación y 5 pulgadas de diámetro.

6. Implementación

En esta sección vamos a explicar el proceso que hemos seguido para desarrollar cada parte del sistema. Como hemos dicho anteriormente, este trabajo es código abierto y todo estará accesible en los siguientes repositorios de Github:

- https://github.com/osmartinez/UAVManager (Aplicación principal de escritorio).
- https://github.com/osmartinez/GestureFlightMode (Biblioteca en C para pilotar mediante gestos).
- https://github.com/osmartinez/KeyboardFlightMode (Biblioteca en C para pilotar mediante teclado).
- https://github.com/osmartinez/UAV (Scripts en Python que se ejecutarán en el vehículo).

6.1. Software para monitorización

Comenzaremos con el software para monitorización por las siguientes razones:

A. Relativamente más sencillo de implementar que el software de pilotaje.
B. Una vez terminado, podremos reutilizar código en el software de pilotaje (sobre todo lo relacionado con el protocolo de comunicación).

Como ya dijimos, en el apartado de selección de tecnologías, en este caso, utilizaremos comunicación mediante sockets UDP.

Para conseguir que el proceso de monitorización se lleve a cabo, se necesita que tanto la Estación, como el vehículo, mantengan una comunicación fluida y constante durante, al menos, el tiempo que dure el vuelo. Para que esta comunicación se lleve a cabo se va a desarrollar un pequeño protocolo de comunicación para ambas máquinas.

En este escenario, tenemos dos nodos (Estación y UAV). Cada uno tiene un rol distinto. Mientras que el UAV se limita a enviar continuamente datos que informan sobre su estado actual, la Estación se limita a recibirlos y mostrarlos por pantalla.

6.1.1. Estación

Ahora que ya sabemos la estructura que va a tener la comunicación y cómo se va a llevar a cabo el paso de mensajes, vamos a recordar las tres características que tiene que ofrecer el programa alojado en la computadora de la Estación:

A. Visualización de modelo 3D que describa la misma orientación que el modelo real.
B. Grabación y visualización del vídeo que retransmite el vehículo.
C. Visualización de la calidad de la señal wifi.

6.1.1.1. Visualización de modelo 3D

Lo primero que necesitaremos es conseguir visualizar un modelo tridimensional de un UAV que simule nuestro vehículo real. Como ya hemos dicho anteriormente, para la visualización de formas 3D, vamos a utilizar el *wrapper* para C# de *OpenGL* denominado *OpenTK*.

Utilizaremos Blender[12] para elaborar el esqueleto de un UAV. Una vez elaborado, lo exportaremos en formato Wavefront OBJ[13]. Hay muchos formatos en los que podemos exportar este modelo, pero tenemos experiencia con este formato en particular por haber trabajado con él durante la asignatura de Prácticas Curriculares.

Sí abrimos este fichero, veremos miles de líneas. Cada línea viene etiquetada por una cabecera:

- v: línea que define un vértice.
- vn: línea que define un vector normal.
- f: línea que define una cara.

Para simplificar el problema y puesto que no es un requisito, no vamos a colorear el modelo, así que no tenemos texturas, solamente vértices, vectores normales y caras. En esencia el fichero es algo como esto:

```
# Comentarios y metadatos
# ...
# Definición de vértices
v -0.607202 2.070846 -0.138455
v -0.616399 2.159815 -0.088970
...
# Definición de vectores normales
vn 0.8819 0.0047 0.4713
vn 0.7688 0.0048 0.6395
...
# Definición de caras.
f 22247//6836 22249//6836 22250//6836
f 22248//6837 22243//6837 22246//6837
...
```

[12] Programa para modelado tridimensional.
[13] Formato de fichero en texto plano que define un objeto tridimensional. Dicha definición incluye vértices, caras, vectores normales y texturas.

El número de líneas de estos ficheros puede variar mucho, dependiendo de la complejidad del modelo, y esto, en ocasiones suele representar un problema en términos de espacio en memoria principal y tiempo de lectura. En nuestro caso, tenemos un fichero de 10.000 líneas y 300 kilobytes.

Los vértices son puntos que se definen como tres coordenadas (x, y, z) en el espacio.

Los vectores normales se definen como tres valores decimales (i, j, k) donde i, j y k son las coordenadas del vector normal. Son utilizados para el sombreado del modelo.

Las caras son primitivas geométricas. Podemos elegir la figura que queramos para representarlas, sin embargo, se suelen utilizar el cuadrado y el triángulo. Principalmente el triángulo porque las GPU trabajan mucho más rápido con estas formas. Vemos que las caras (triangulares) son representadas por tres componentes (v//vn v'//vn' v''//vn''). Es importante apreciar que el orden en que se han escrito las líneas importa, porque las caras hacen referencia a índices en que se encuentran los vértices y vectores normales.

Una vez hemos entendido cómo se define un modelo tridimensional en este formato, vamos a cargarlo en memoria principal de manera que OpenGL pueda interpretarlo y mostrarlo por pantalla.

Las funciones necesarias que hemos desarrollado para lograrlo se encuentran en el siguiente fichero:

https://github.com/osmartinez/UAVManager/blob/master/CargadorOBJ/OBJ.cs

En el fichero podemos ver dos funciones:

Función	**Descripción**	**Parámetros**	**Devuelve**
ReadOBJ	Lee los vértices, vectores normales y caras del fichero y los almacena en memoria.	**fichero**: cadena de caracteres que indica la ruta del fichero OBJ. **cambiarYZ**: booleano que indica si debemos voltear el modelo.	**Verdad**: si no ha habido ningún problema durante la lectura. **Falso**: en caso contrario.
ArmList	Genera y compila una lista de comandos OpenGL que esperan ser ejecutados para visualizar el modelo 3D.		**Verdad**: si no ha habido ningún problema durante el generado de la lista. **Falso**: en caso contrario.

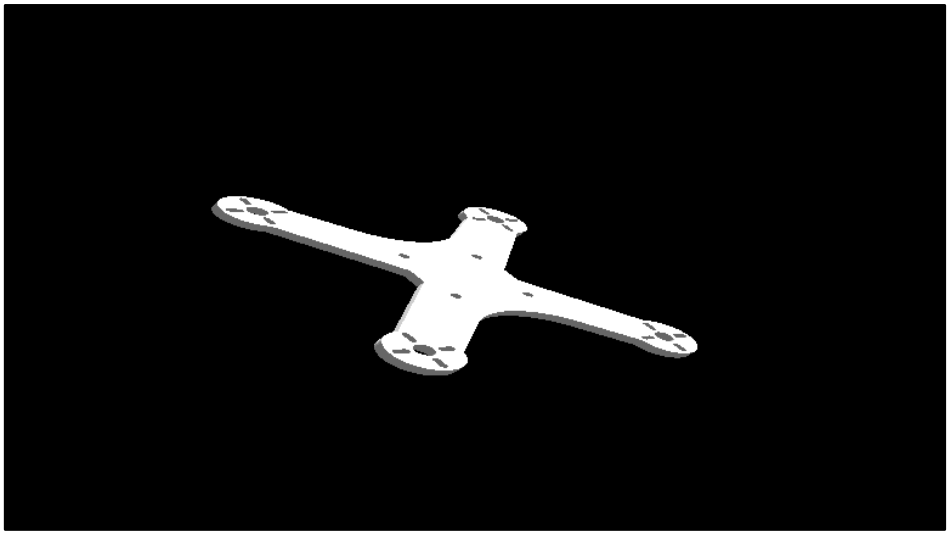

Figura 4. Resultados obtenidos en visualización del modelo.

Tras las funciones anteriormente explicadas para cargar nuestro modelo y visualizarlo obtenemos el resultado que se muestra en la figura 4.

Ahora necesitamos desarrollar una función encargada de extraer los valores (descriptores de la orientación) que nos está enviando el vehículo (utilizando sockets UDP). Para ello necesitaremos establecer a priori un formato en el que van a viajar los datos para que tanto UAV como Estación puedan entenderse.

El formato en el que viajen los valores de la orientación del vehículo serán los siguientes:

{'angx':angx, 'angy':angy, 'heading':heading}

Estos valores nos darán la orientación del vehículo en sus tres grados de libertad.

La función encargada de obtener estos valores se encuentra en el fichero accesible desde:

https://github.com/osmartinez/UAVManager/blob/master/Communication/Monitoring.cs

En particular, la función tiene la siguiente especificación:

Función	Descripción	Parámetros	Devuelve
ReceiveAngles	Pone a la escucha un socket UDP para recibir el paquete del servidor. Trocea el mensaje y extrae los ángulos, para almacenarlos en el objeto activo.		

Tenemos nuestro modelo cargado y visualizado, y tenemos la función que obtiene los valores de la orientación del vehículo en tiempo real. Así que vamos a utilizar los valores obtenidos para aplicar las transformaciones necesarias al modelo tridimensional.

Por lo general, se usan matrices para computación gráfica porque podemos concatenar fácilmente operaciones de traslación, reflexión, rotación, escalado, etc.

Para conseguir el objetivo, necesitaremos un ciclo, y en cada iteración aplicar las tres rotaciones al modelo en los tres ejes, con los valores en grados que hemos obtenido del vehículo.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 5. Matriz inicial de transformación

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figura 6. Vector de cada eje del modelo.

Cada eje del modelo se representa por un vector como el de la figura 6. El proceso de transformación del modelo sigue las siguientes etapas:

1. Cargar matriz identidad del modelo.
2. Aplicar rotación en *x*.
3. Aplicar rotación en *y*.
4. Aplicar rotación en *z*.

Las transformaciones que realizaremos son únicamente rotaciones, puesto que el modelo quedará fijo en el origen.

La explicación de por qué se usan matrices para computación gráfica es porque podemos concatenar fácilmente operaciones de traslación, reflexión, rotación, escalado, etc.

El motivo por el cual usamos matrices 4x4, y no 3x3, cuando trabajamos con gráficos en 3 dimensiones, es porque las matrices solamente pueden modelar transformaciones o aplicaciones lineales. Este hecho nos limita un poco, ya que, aunque podemos representar rotaciones, con matrices 3x3 no podemos representar traslaciones (transformaciones afines).

Las transformaciones lineales se aplican respecto a un origen, dejándolo fijo. Es por la propia naturaleza de una traslación (desplazar el origen) que no puede ser representada por una aplicación lineal. La solución viene dada por la ampliación de la matriz en una dimensión más.

Las rotaciones se realizan por separado en cada eje y se pueden concatenar. El objetivo es encontrar el vector de la transformación que, aplicado a nuestro modelo, provoque un giro de θ grados alrededor del eje esperado.

Para cada eje se aplica una matriz distinta de rotación. Para el eje *x* tenemos la siguiente:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para el eje *y* tenemos la siguiente matriz:

$$\begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Y por último la matriz para el eje *z*:

$$\begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Tomamos ahora cada matriz y la multiplicamos al vector de su eje correspondiente. De esta forma, obtenemos el siguiente vector para el eje *x*:

$$\begin{bmatrix} x \\ ycos\theta - zcos\theta \\ ysin\theta + zcos\theta \\ 1 \end{bmatrix}$$

El siguiente para el eje *y*:

$$\begin{bmatrix} xcos\theta + zsin\theta \\ y \\ -xsin\theta + zcos\theta \\ 1 \end{bmatrix}$$

Y por último el vector para el eje *z*:

$$\begin{bmatrix} xcos\theta - ysin\theta \\ xsin\theta + ycos\theta \\ z \\ 1 \end{bmatrix}$$

Tras obtener los vectores resultantes, éstos son aplicados al modelo con lo que podremos visualizar el nuevo modelo transformado.

Se puede observar cómo aplicamos dichas transformaciones en la definición del manejador glc_Paint que se encuentra aquí:

https://github.com/osmartinez/UAVManager/blob/master/UAVManager/ManagerFrm.cs

6.1.1.2. Grabación y visualización de vídeo

Primeramente, vamos a abordar la visualización en tiempo real del vídeo retransmitido por la cámara del vehículo.

Como hemos especificado anteriormente, vamos a utilizar *EmguCV*, un *wrapper* de C# para *OpenCV*, para visualizar las imágenes que nos está enviando continuamente el vehículo. El mayor problema al que nos enfrentaremos, es la conversión de tipos que implica enviar un objeto desde Python hasta C# por red, en particular, enviar una matriz de dimensión 640x480 en forma de bytes y convertirla a un objeto *EmguCV* para poder trabajar él y visualizarlo.

El proceso para conseguir el objetivo deseado es el siguiente:

1. Obtener los datos de la última imagen que nos envía el vehículo.
2. Transformar dichos datos a un objeto C# con el que podamos operar.
3. Pintar dicha imagen sobre un control de tipo *Picturebox* de *Windows Forms*.

En el fichero accesible desde aquí:

https://github.com/osmartinez/UAVManager/blob/master/Communication/Monitoring.cs

podemos encontrar la función *ReceiveFrame* encargada de realizar los pasos 1 y 2 del proceso previamente descrito. Durante el desarrollo de la función hemos tenido varios problemas relacionados con la conversión de tipos como explicamos anteriormente.

Se buscó ayuda en el sitio web de StackOverflow. Desde el perfil del estudiante, creamos una pregunta en la que se describe claramente el problema y el resultado que se quiere obtener. Finalmente, tras no recibir respuesta, es el estudiante quien encuentra una solución. La pregunta y la solución están accesibles desde el siguiente enlace:

https://stackoverflow.com/questions/49906779/udp-sending-frames-from-OpenCV-to-EmguCV/

Ahora necesitamos poder salvar (si el usuario lo desea), el vídeo que está siendo transmitido, al disco. Para ello, necesitaremos crear un fichero de vídeo en la ruta escogida por el usuario. La nomenclatura de los archivos de vídeo generados va a venir dada por la fecha y hora en la que se crea el fichero, de manera que el usuario pueda identificarlos fácilmente en caso que haya grabado numerosos vídeos.

Haremos uso de la clase VideoWriter que nos proporciona EmguCV, y cada vez que mostremos el frame recibido por pantalla, lo volcaremos al archivo de vídeo que esté abierto en ese momento.

Finalmente, al acabar la transmisión, cerramos el stream y el vídeo habrá quedado grabado.

6.1.1.3. Visualización de calidad de la señal

Para visualizar la calidad de la señal wifi que le está llegando al vehículo necesitamos seguir estos dos pasos:

1. Obtener la información que nos está enviando el vehículo sobre la red.
2. Graficar en tiempo real la calidad de la señal.

Al igual que en caso de los datos sobre la orientación del vehículo, ahora también necesitamos establecer un formato para el paquete que va a contener la información. Su formato será el siguiente:

{<nombreDeLaRed> <canal> <calidad> <intensidad>}

En el fichero:

https://github.com/osmartinez/UAVManager/blob/master/Communication/Monitoring.cs

existe una función llamada *ReceiveWifiData* que desempeña la funcionalidad del paso 1 del proceso.

Función	Descripción	Parámetros	Devuelve
ObtenerDatosWifi	Pone a la escucha un socket UDP para recibir el paquete del servidor. Trocea el mensaje y extrae toda la información de la red para almacenarla en memoria.		

Ya solo queda integrar en el código de la interfaz principal un ciclo que lea la calidad de la señal y la grafique continuamente. Crearemos una tarea que corra en paralelo y que vaya accediendo a los valores recogidos del socket.

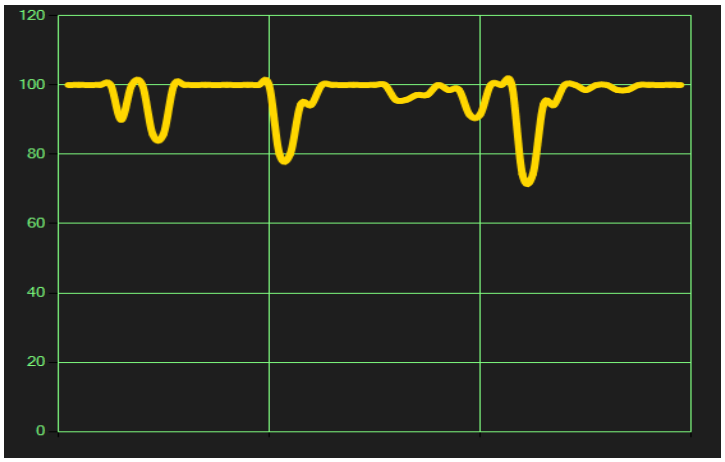

Figura 7. Resultados obtenidos tras la visualización de calidad de la señal.

6.1.2. UAV

6.1.2.1. Envío de datos IMU

Vamos a abordar el envío de información relacionada con la orientación del vehículo. Estos datos los proporcionan los diferentes sensores que están integrados en la placa (acelerómetro, giroscopio, barómetro, magnetómetro, etc.).

La obtención de toda esta información la realizamos mediante un protocolo en serie denominado MSP (Multiwii Serial Protocol). Para ello, utilizaremos un módulo en Python ya existente (pyMultiwii) que nos proporciona una serie de funciones para conectarnos y extraer los datos que necesitamos de la placa controladora de vuelo.

Nuestro módulo Python tiene que realizar el siguiente proceso:

1. Conectar con la placa.
2. Obtener datos IMU.
3. Enviarlos a la Estación.

El fichero que proporciona dicha funcionalidad está accesible desde:

https://github.com/osmartinez/UAV/blob/master/UAVScripts/UAVScripts/monitor.py

En dicho fichero podemos ver una clase denominada *Monitor* cuyo constructor recibe la IP de la computadora de la Estación y el puerto para la comunicación. Los objetos de dicha clase contienen un método denominado *run* que tiene la siguiente especificación:

Función	Descripción	Parámetros	Devuelve
run	Conecta con la placa, y mientras la conexión siga en pie, recoge la información de la placa y la envía a la Estación vía UDP.		

6.1.2.2. Envío de vídeo

Como ya hemos dicho anteriormente, para recoger el vídeo de la cámara del vehículo vamos a utilizar la librería OpenCV para Python.

El proceso de envío de la imagen es algo más complejo que los anteriores:

1. Conectar con la cámara.
2. Establecer calidad o formato de la imagen.
3. Establecer una resolución.
4. Recoger imagen de la cámara.
5. Codificar la imagen.
6. Enviar la imagen codificada.

El fichero donde se puede encontrar toda la funcionalidad es el siguiente:

https://github.com/osmartinez/UAV/blob/master/UAVScripts/UAVScripts/camraspi.py

Como en el apartado anterior, tenemos una clase. En este caso, la clase se denomina Cam y tiene la misma estructura que Monitor. En el constructor establecemos la resolución y calidad de la imagen. Y en el método run, mediante un bucle, vamos obteniendo las imágenes de la cámara y enviándolas siguiendo el proceso anteriormente descrito.

Como extra, hemos añadido otra clase que más sencilla preparada para funcionar con una cámara conectada por USB, por si en un futuro funcionamos con otra cámara de más prestaciones. Dicha clase está accesible en el siguiente fichero:

https://github.com/osmartinez/UAV/blob/master/UAVScripts/UAVScripts/camusb.py

6.1.2.3. Envío de información sobre la conexión

Para implementar esta funcionalidad, no vamos utilizar ningún tipo de biblioteca externa para acceder a la información que nos proporciona la tarjeta de red. Esto es porque el propio sistema operativo (Raspbian) nos ofrece un conjunto de órdenes muy útiles que ya nos proporcionan la información que necesitamos.

En concreto, el comando necesario para ver dicha información es “iwlist”. Así que abriremos un subproceso y ejecutaremos ese comando. Obtendremos el resultado de la ejecución en forma de cadena de caracteres y extraeremos la información que necesitamos. Una vez que

tenemos toda esa información, elaboraremos el mensaje con los datos que van a viajar a la Estación.

El desarrollo de esta funcionalidad se puede ver en el siguiente fichero:

https://github.com/osmartinez/UAV/blob/master/UAVScripts/UAVScripts/wifi_monitor.py

6.2. Software para pilotaje

A continuación, veremos el proceso de desarrollo del software para pilotar el vehículo. Explicaremos primero el software que irá desplegado en la Estación, que es más complejo, y finalizaremos con la explicación del proceso de desarrollo del programa encargado de ejecutar las órdenes generadas en la Estación.

Los cuadricópteros, al igual que cualquier vehículo aéreo se mueven con tres ángulos de libertad. Estos ángulos, son denominados Pitch, Yaw y Roll. Estos ángulos son también conocidos como ángulos de navegación y a su vez, pueden ser útiles para determinar la orientación de un objeto.

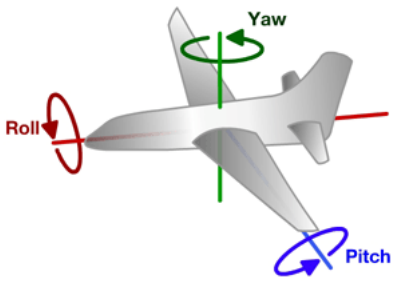

Figura 8. Representación de los ángulos de navegación en vehículo aéreo.

En profundidad:

- Pitch: hace referencia al movimiento del vehículo cuando éste se gira sobre el eje que lo atraviesa longitudinalmente. Esto produce un movimiento hacia delante en el caso de que la inclinación sea positiva, o hacia atrás en caso contrario. Para dar lugar a este giro, el vehículo reduce la velocidad de los motores traseros y aumenta los delanteros o viceversa.
- Yaw: se refiere al movimiento que describe el vehículo al girar sobre su vertical. El vehículo reduce o aumenta la velocidad de giro de los motores en diagonal para conseguir un giro a izquierdas o derechas.
- Roll: el vehículo gira sobre el eje que lo atraviesa lateralmente. Tiene la misma explicación que para el Pitch, pero en este caso agrupamos los pares de motores según estén a izquierda o derecha.

Notar que tanto Pitch como Roll producen desplazamiento. A diferencia de Yaw, que no produce más que un movimiento del vehículo sobre su eje vertical.

Además, también tendremos que tener en cuenta un movimiento importante: Throttle. El Throttle, rige la velocidad de los motores, y es lo que marca la altura a la que estamos volando.

Debemos considerar un detalle antes de desarrollar los programas de pilotaje: la escala en la que se mueven los valores de los ángulos de navegación y Throttle.

Para Yaw, Pitch y Roll, el valor medio, es decir el valor en el que el vehículo no realiza ningún tipo de giro es 1500. De manera que, si aumentamos ese valor, el vehículo realizará un giro con ángulo positivo, y si lo disminuimos, el vehículo realizará un giro con ángulo negativo.

Por otra parte, para el Throttle funciona algo distinto, puesto que los valores van de 1000 a 2000, de manera que 1000 refleja la aceleración nula y 2000 la máxima velocidad que alcanzan los motores.

La magnitud con la que podremos variar estos valores irá ajustándose mediante prueba y error cuando por fin volemos el UAV.

6.2.1. Estación

6.2.1.1. Pilotaje mediante teclado

Como hemos dicho anteriormente, este módulo va a ser desarrollado en C# utilizando Windows Forms para diseñar la interfaz. Buscamos una interfaz sencilla y manejada íntegramente por teclado.

El programa debe ofrecer la funcionalidad suficiente como para comandar el vehículo pudiendo controlar los tres ángulos de navegación y además la velocidad de los motores (Throttle). Pero a su vez ser muy sencillo sin perder usabilidad.

Una idea que creemos acertada es representar los valores de cada parámetro en forma gráfica mediante barras de progreso, de esta manera el usuario no tendrá que fijarse en el valor numérico. Además, también será útil identificar cada barra con el parámetro al que hace referencia.

Ahora nos debemos centrar en las combinaciones de teclado que vamos a utilizar para controlar estos 4 parámetros.

Deberemos mapear 4 pares de teclas para controlar las dos posibles variaciones (incremento y decremento) de cada parámetro. Además, necesitaremos una tecla más para indicar que finalizamos el pilotaje o que hubo algún fallo (emergencia) y necesitamos que el vehículo se desarme.

Figura 9. Teclas utilizadas.

Seguiremos la aproximación tradicional que usan muchos emuladores o juegos de ordenador que utilizan teclado para el manejo, donde las teclas que se reservan para el movimiento suelen ser: W,S,A,D y las teclas de dirección (figura 9). Utilizaremos la tecla de Escape para indicar parada de emergencia o fin del pilotaje.

Ahora mapearemos cada par de teclas con cada parámetro:

Parámetro	**Tecla para incrementar**	**Tecla para decrementar**
Throttle	W	S
Yaw	A	D
Pitch	UP	DOWN
Roll	RIGHT	LEFT
Emergencia	ESC	

6.2.1.2. Pilotaje mediante gestos

Para realizar este módulo de pilotaje, vamos a utilizar el lenguaje C++ junto con la SDK de OpenNI para acceder a las funciones de "skeletal tracking" del sensor Kinect como ya explicamos anteriormente.

En este caso, a diferencia del modo de pilotaje por teclado, no necesitaremos una interfaz de usuario como tal, porque el usuario se va a encontrar normalmente a cierta distancia de la Estación. Solo le interesa ser captado por la cámara mientras mantiene a la vista el vehículo.

Igual que en el caso anterior, debemos encontrar un set de movimientos que puedan ser mapeados a órdenes de pilotaje para el vehículo. Sin embargo, ahora la complejidad se ve aumentada debido a que el cuerpo humano tiene mucha más libertad de movimiento que la que nos puede ofrecer un teclado. Por ello, tendremos que delimitar correctamente los gestos que entrarán dentro de nuestro conjunto de movimientos válidos.

Con el fin de hacer la explicación más llevadera, es necesario aclarar dos puntos.

Primero vamos a elegir un foco (espectador) que usaremos como punto de vista para explicar este módulo. A la hora de desarrollar, vamos a tener que posicionarnos en el lugar donde está el sensor, así que nuestro foco o punto de vista coincidirá con las mismas coordenadas del sensor en el espacio. El ángulo de visión también estará limitado por el que nos ofrezca el sensor.

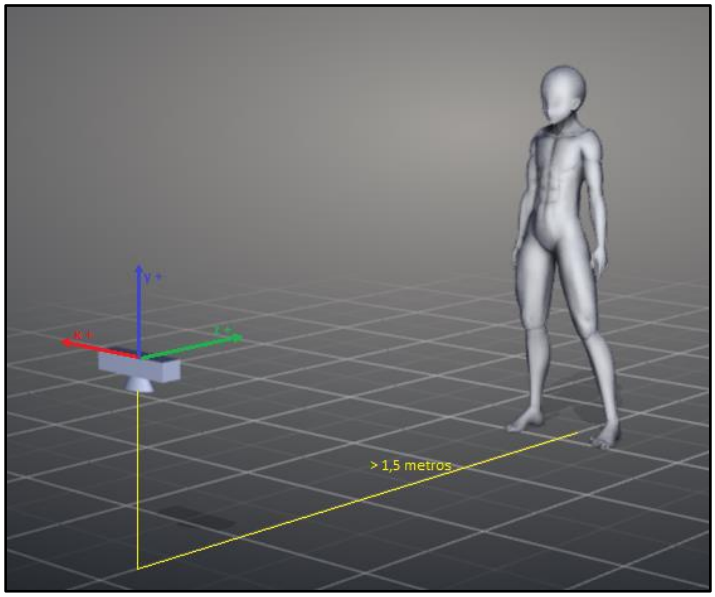

Figura 10. Disposición del escenario del sistema para módulo de pilotaje basado en gestos.

Y en segundo lugar, vamos a situar al actor del sistema y nombrar los ejes del espacio. El usuario debe situarse en frente del sensor, a una distancia superior a 1,5 metros (como indica Microsoft). El origen de coordenadas está situado en la lente de nuestro sensor.

Tanto la dirección y nomenclatura de los ejes de coordenadas como la posición del usuario respecto al sensor pueden verse apreciadas en la figura 10.

Teniendo ya una visión general sobre el escenario de nuestro sistema es hora de generar nuestro conjunto de gestos válidos que serán mapeados a órdenes de pilotaje.

Ya vimos en la introducción del apartado los grados de libertad que tiene el vehículo. Ahora necesitaremos encontrar gestos que vayan acorde con ellos. En el segundo anexo estudiaremos mucho más a fondo la descripción de estos gestos. Aquí simplemente diremos cuáles son:

Valor	Incremento	Decremento	Brazos
Throttle	desplazar arriba	desplazar abajo	izquierdo
Yaw	desplazar izquierda	desplazar derecha	izquierdo
Pitch	desplazar arriba	desplazar abajo	derecho
Roll	desplazar izquierda	desplazar derecha	derecho

Emergencia			izquierdo y derecho

Ahora que ya hemos mapeado cada gesto con su valor, vamos a profundizar en los cálculos necesarios para generar la orden asociada a un gesto. Veremos que para el cálculo de los valores asociados utilizaremos un recurso denominado triangulación, es decir, generaremos triángulos en el espacio y mediante trigonometría, conseguiremos obtener distancias y ángulos necesarios para nuestros cálculos.

Throttle.

El brazo encargado de regular este valor es el izquierdo. El gesto consiste en bajar o subir la mano dejando (en la medida de lo posible) el codo fijo, para suministrar más o menos velocidad a los motores. La propia naturaleza del gesto hace que podamos prescindir de una coordenada (x), puesto que solo nos va a interesar la posición de las articulaciones en los ejes z e y.

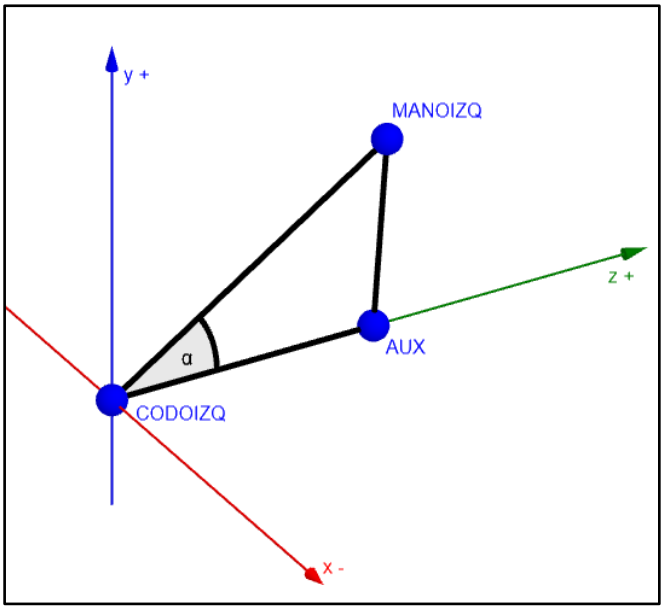

Figura 11. Cálculo de Throttle mediante triangulación de codo y mano.

Tomaremos un triángulo cuyos puntos son determinados por:

- Coordenadas de la mano izquierda: sin tomar la coordenada en x.
- Coordenadas del codo izquierdo: sin tomar la coordenada en x.
- Auxiliar: punto calculado al combinar la coordenada y del codo con la coordenada z de la mano. El valor de las x también será eliminado.

Con los tres puntos, podemos calcular α sin problema. Es importante tener en cuenta el signo del ángulo. Para empezar, calcularemos la diferencia de altura entre las coordenadas y de codo y mano. De esta manera, si la mano está más alta que el codo, sabremos que el ángulo será positivo, y negativo en caso contrario.

Una vez sabemos el valor que toma α, necesitaremos convertirlo a un valor de Throttle válido, puesto que como dijimos, este valor tiene un rango de [1000, 2000]. Para transformar α a un valor de Throttle válido utilizaremos la siguiente fórmula que hemos desarrollado:

$$t = \frac{\alpha \cdot 1000}{c} + 1500$$

Dónde α es el ángulo calculado y c es la cota o límite de apertura del brazo en grados.

Es necesario aclarar que, por motivos relacionados con la anatomía del cuerpo humano, que impide llegar a formar 90º con la mano por debajo del codo, hemos decidido acotar el ángulo calculado en ±80º. Hemos observado que 80, en nuestro caso de uso, genera buenos resultados y por ello escogemos este valor. En total, la apertura sería de 160º (el doble), así que tenemos que c=160.

Yaw.

Igual que en el caso anterior, será el brazo izquierdo el encargado de regular este valor. Sin embargo, en éste, solo tendremos en cuenta el desplazamiento del brazo en la dimensión x.

También vamos a calcularlo mediante triangulación. En este caso y en los que quedan, solamente nos interesa saber hacia qué lado está apuntando la mano porque estableceremos una velocidad de giro predeterminada y simplemente giraremos en una dirección u otra dependiendo en la posición del brazo.

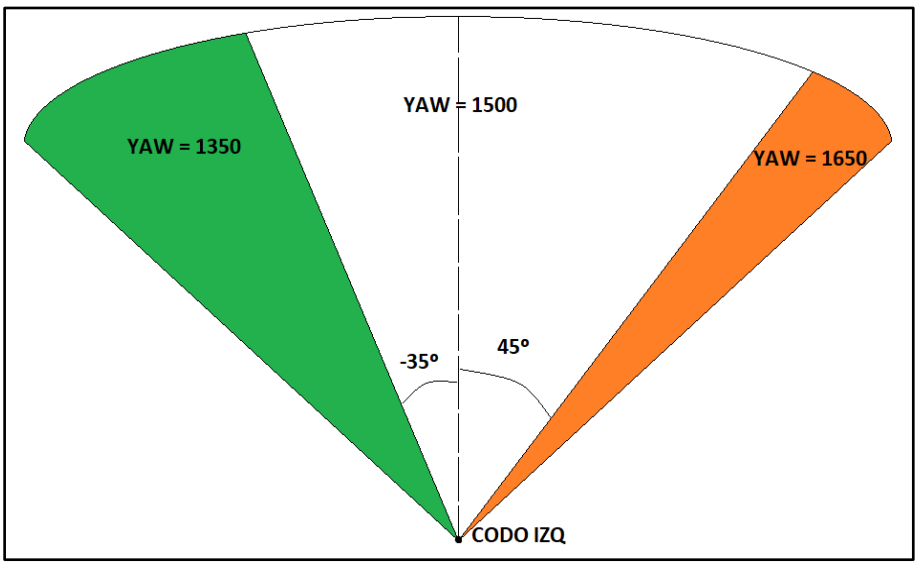

Figura 12. Distribución de regiones para controlar Yaw.

Es importante tener en cuenta que, a un ser humano, en general, le cuesta mantener constante la posición de sus articulaciones. Es por ello que tendremos que establecer unas ventanas que resuelvan este problema y las posibles pequeñas variaciones de movimientos que se puedan producir cuando el brazo se encuentra en el límite de estas ventanas.

Además, también hemos observado que por la propia anatomía del brazo, resulta más fácil y cómodo flexionarlo (+Yaw), que estirarlo (-Yaw), por ello, como se observa en la figura 12, hemos hecho que el programa sea algo más sensible cuando se produzca el gesto de estirar el brazo. En cuanto a la región del centro, simplemente hay que decir que es la región de descanso o de Yaw nulo, es decir, el vehículo permanece en la orientación que estaba.

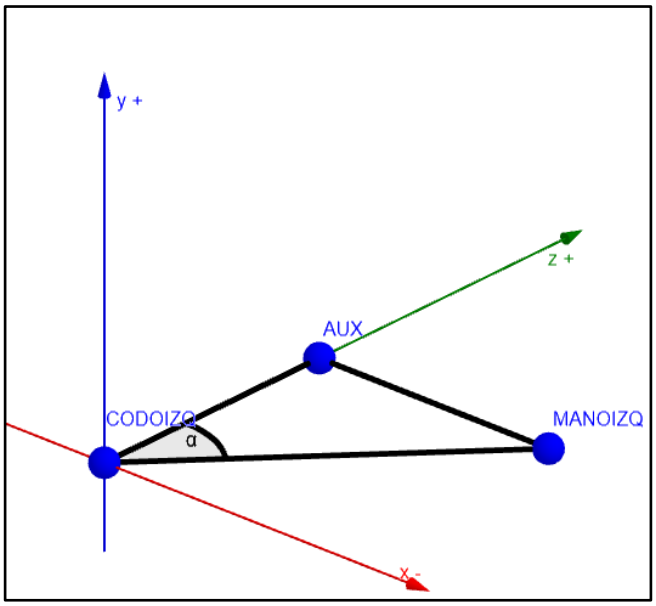

Figura 13. Cálculo de Yaw mediante triangulación de mano y codo.

Para calcular el ángulo del triángulo generado, en este caso, vamos a prescindir de la coordenada y, y haremos los cálculos simplemente con *x* y *z*. El triángulo estará formado por los siguientes puntos:

- Coordenadas de la mano: sin tomar la coordenada en *y*.
- Coordenadas del codo: sin tomar la coordenada en *y*.
- Auxiliar: punto calculado al combinar la coordenada z de la mano con la coordenada x del codo. El valor en el eje y tampoco se tendrá en cuenta.

Supongamos que cuando la mano está alineada con el codo en el eje x, α toma valor 0º. Así mismo, cuando la mano se desplaza a derechas, α se ve incrementada y al revés si la mano se desplaza a izquierdas. Teniendo en cuenta esto y la representación realizada en la figura 13, el valor del Yaw vendrá dado por la siguiente función:

$$f(\alpha) = \begin{cases} 1650 & : \alpha > 45 \\ 1350 & : \alpha < -35 \\ 1500 & : \text{en caso contrario} \end{cases}$$

Estos valores (1650 y 1350) han sido elegidos por proporcionar una buena relación desplazamiento-agresividad. En otros sistemas, estos valores deberían ser estudiados y elegidos mediante algún procedimiento de prueba-error. Aun así, en nuestro programa, podremos cambiar estos valores en el módulo de configuración.

Pitch.

En este caso, vamos a tomar el mismo gesto que para Throttle pero con el brazo derecho. De manera que al aumentar la altura de la mano con respecto al codo haremos que el vehículo se incline hacia atrás y con ello retroceda, y de otra forma, si decrementamos la altura de la mano haremos que el UAV se incline y desplace hacia delante.

A diferencia del Throttle, y al igual que con el Yaw, no vamos a establecer una equivalencia entre los grados de inclinación de nuestro brazo y la inclinación del vehículo.

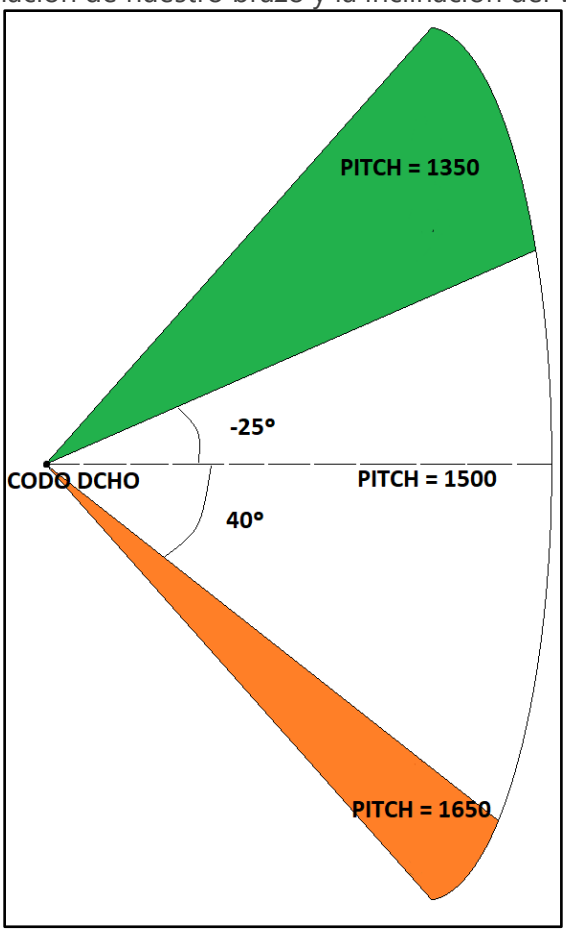

Figura 14. Distribución de regiones para calcular Pitch.

Simplemente vamos a detectar si el gesto está pidiendo que el vehículo avance, o que por el contrario retroceda. De esta forma, daremos un valor fijo de inclinación y lo aplicaremos en un sentido u otro dependiendo del gesto detectado.

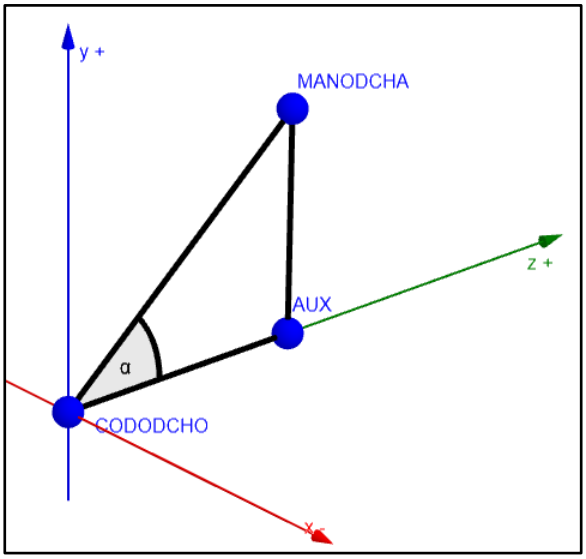

Figura 15. Cálculo de Pitch mediante triangulación de mano y codo derechos.

El triángulo que calcularemos estará compuesto por los siguientes puntos:

- Coordenadas de la mano derecha: sin tomar la coordenada en *x*.
- Coordenadas del codo derecho: sin tomar la coordenada en *x*.
- Auxiliar: punto calculado al combinar la coordenada *y* del codo con la coordenada *z* de la mano. El valor de las *x* también será suprimido.

Roll.

Tomaremos el mismo gesto que para Yaw pero con el brazo derecho. De nuevo, no estableceremos una relación entre el ángulo calculado y el valor de Roll, sino que utilizaremos un valor fijo que será aplicado en un sentido u otro dependiendo del gesto.

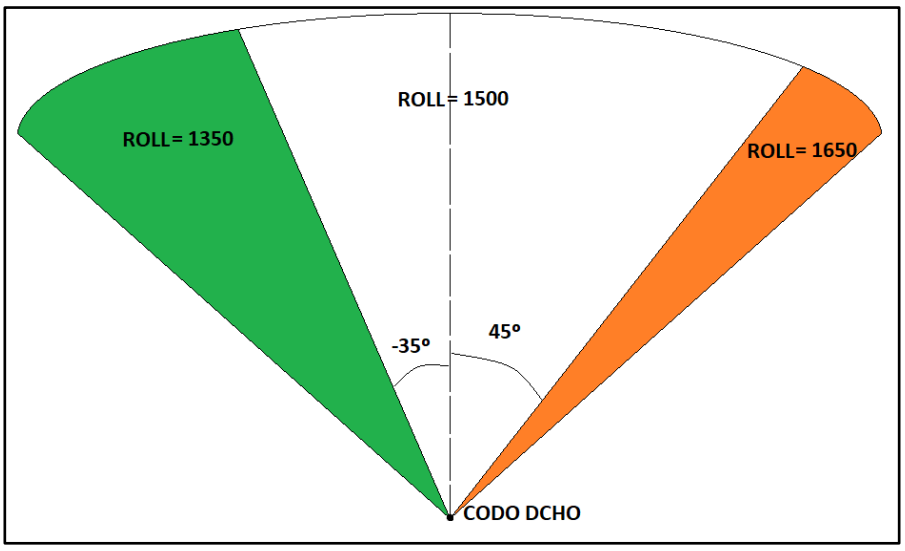

Figura 16. Distribución de regiones para calcular Roll.

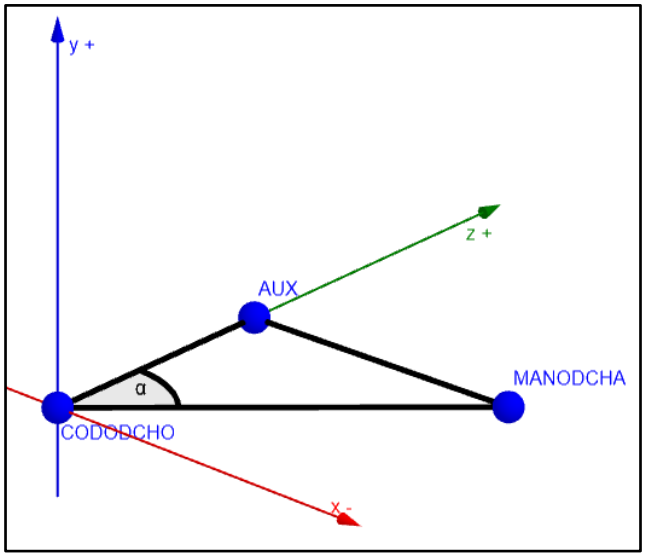

Figura 17. Cálculo de Roll mediante triangulación de mano y codo derechos.

Como apreciamos en la figura 17, el triángulo que se forma está compuesto por:

- Coordenadas de la mano derecha, sin tomar la coordenada en *y*.
- Coordenadas del codo derecho sin tomar la coordenada en *y*.
- Punto auxiliar calculado mediante la componente en *z* de la mano derecha y la componente *x* del codo derecho.

Emergencia.

Para ejecutar este comando, utilizaremos ambos brazos. El gesto consiste en juntar las dos manos. La manera que tenemos de comprobar si este gesto se ha producido es calculando la distancia euclídea entre los puntos que posicionan las muñecas de ambos brazos. Sí el resultado de esta distancia es menor que un determinado valor (100, por prueba-error), consideraremos que se ha llevado a cabo el gesto y mandaremos una orden de parada al vehículo.

6.2.2. UAV

El módulo encargado de pilotaje en el lado del vehículo es mucho más sencillo que el de la estación. Solo deberemos realizar un script en Python que se encargue de:

1. Recibir la orden de movimiento.
2. Trocear la orden y sacar los valores.
3. Enviar dichos valores a la controladora de vuelo mediante protocolo MSP para que los ejecute.

A continuación, se presenta un diagrama de actividad para explicar dicho proceso:

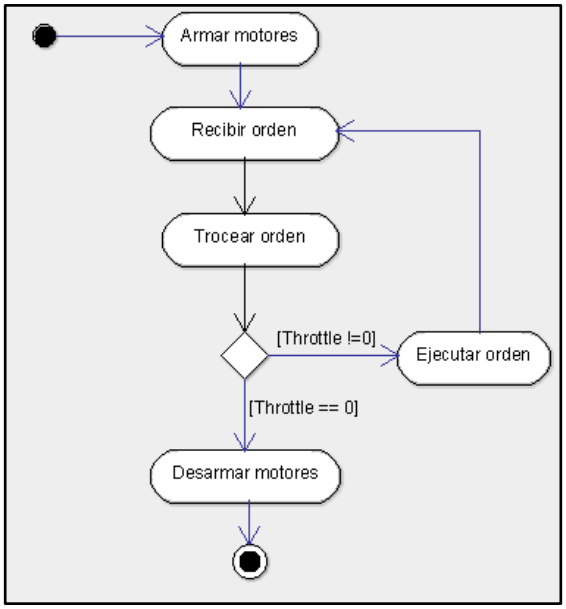

Figura 18. Flujo de programa para el módulo Python encargado de ejecutar órdenes.

Como podemos ver, nuestro código de desarme de motores consiste en escribir el valor 0 en la componente Throttle, de esta manera nuestro módulo Python puede detectar que ha habido algún problema o que simplemente queremos aterrizar y él mismo se encarga de desarmar los motores del vehículo.

El script que contiene dicha funcionalidad está accesible desde:

https://github.com/osmartinez/UAV/blob/master/UAVScripts/UAVScripts/flyTCP.py

6.3. Configuración y ensamblaje de UAV

El primer procedimiento que se sigue a la hora de implementar el vehículo consiste en configurar la Raspberry Pi. Ésta tiene instalado el sistema operativo Raspbian (distribución Linux basada en Debian). Necesitaremos instalar los intérpretes, bibliotecas y compiladores necesarios para poder ejecutar nuestros programas.

También necesitaremos configurar la tarjeta de red para que el vehículo se conecte correctamente a nuestra red de manera automática al iniciar el sistema operativo.

Por último, crearemos los scripts en *bash* necesarios que ejecutarán nuestros programas una vez se haya arrancado el sistema.

Instalaremos el intérprete de Python 2.7 con las siguientes bibliotecas:

- pyMultiwii: para acceder a la información que nos proporcionan los sensores de la controladora de vuelo y poder enviarle órdenes a través de protocolo MSP.
- OpenCV: para capturar frames de la videocámara.

Una vez completada la configuración del computador a bordo del vehículo, comenzamos con el ensamblaje de los componentes.

Para ello, lo primero soldaremos los motores con los controladores de velocidad y los montaremos sobre los soportes del chasis.

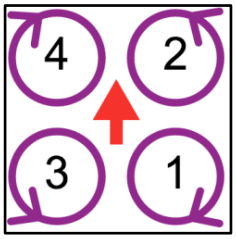

Figura 19. Sentido de giro de los motores.

Se ha prestado especial atención a la hora de soldar los motores, pues se han invertido el sentido de dos de los cables para que giren en la dirección correcta como se puede ver en la figura 19. Los motores en las posiciones 1 y 4 deben girar en sentido horario y los de las posiciones 2 y 3 en sentido antihorario.

También es necesario que la controladora de vuelo se encuentre en el centro de gravedad del vehículo.

Para ubicar la batería y la Raspberry Pi, que son los objetos más pesados, veremos cuáles son los posibles lugares. Estando el interior del chasis ocupado por el cableado y la controladora de vuelo, solo nos quedan disponibles la parte alta y baja del mismo.

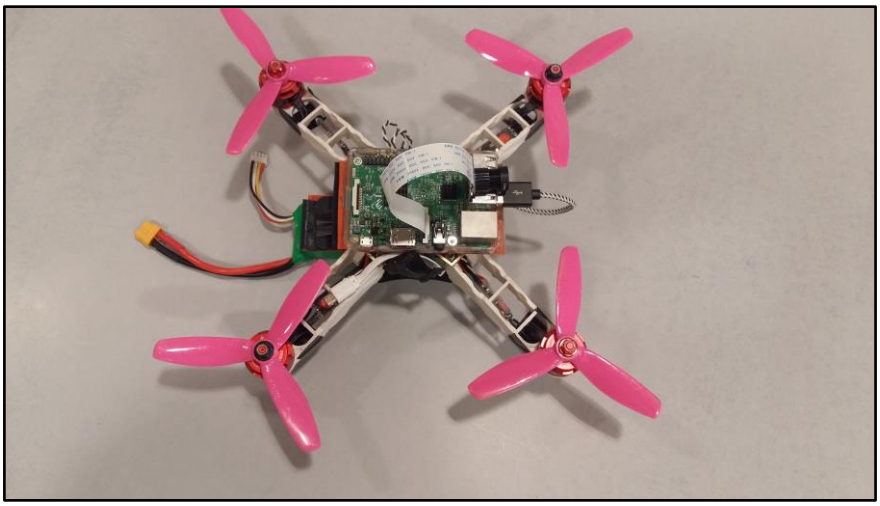

Figura 20. UAV completamente armado.

La parte alta está menos expuesta a golpes, por ello, la Raspberry irá localizada ahí, y por contra, la batería irá ubicada debajo del chasis. Además, vamos a utilizar una carcasa para proteger aún más el computador.

7. Pruebas

Una vez finalizada la fase de implementación, nos disponemos a realizar las pertinentes pruebas con el fin de comprobar la estabilidad y correcto funcionamiento del sistema.

Dividiremos esta fase en dos tareas:

1. Realización de tests unitarios: Comprobaremos unitariamente los métodos más importantes o críticos del sistema. El fin es detectar cualquier tipo de error o falta de robustez en el comportamiento individual de los métodos.

2. Realización de pruebas de integración: Haremos un simulacro de ejecución que involucre todas las partes del sistema, de esta manera comprobaremos cómo se comportan los módulos al interactuar como conjunto.

7.1. Tests unitarios

Para llevar a cabo esta fase, utilizaremos MS Test, un framework que viene integrado en Visual Studio para la ejecución de test unitarios en la plataforma .NET.

7.1.1. Software para monitorización

Comenzaremos con verificar el funcionamiento unitario de los métodos encargados de leer y cargar en memoria el modelo tridimensional de nuestro dron.

La clase que contiene dichos tests se denomina *ReadWriteOBJConfigurationTest* y se puede encontrar en el siguiente enlace:

https://github.com/osmartinez/UAVManager/blob/master/TestConfiguration/ReadWriteOBJConfigurationTest.cs

7.1.2. Módulo de configuración

Vamos a seguir ahora con los tests sobre el proyecto que contiene la funcionalidad de la persistencia para la configuración de la aplicación. Las clases que contiene dichos tests se denominan *ReadWritePilotingValuesConfigurationTest* y *ReadWriteVideoConfigurationTest*. Se pueden encontrar en el siguiente enlace:

https://github.com/osmartinez/UAVManager/tree/master/TestConfiguration

En él podemos ver todo tipo de test que comprueban la lectura y escritura de los parámetros de configuración del programa.

7.1.3. Software para pilotaje

Finalmente, vamos a comprobar el módulo más importante, el de pilotaje. Como ya dijimos anteriormente, en este proyecto importamos bibliotecas DLL que hemos desarrollado en C para utilizarlas en nuestro programa de C#. Por ello, es importante que no dejemos de verificar el correcto funcionamiento de todos estos métodos importados.

El proyecto que contiene los tests para el software de pilotaje se denomina TestPiloting y está accesible desde:

https://github.com/osmartinez/UAVManager/tree/master/TestPiloting

7.2. Pruebas de integración

Una vez hemos pasado todos los tests unitarios, es hora de ejecutar el sistema y comprobar su funcionamiento.

Como ya hemos visto, hemos desarrollado un sistema de configuración con persistencia mediante ficheros XML, lo cual ha dado como resultado un programa mucho más flexible.

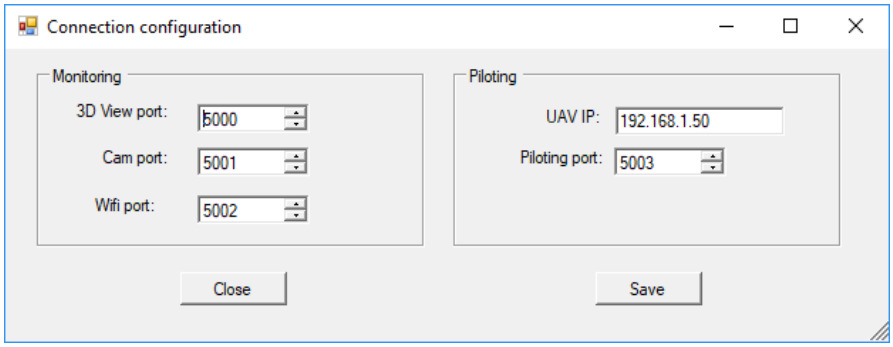

Figura 21. Valores de configuración de la conexión para fase de pruebas.

Para poder llevar a cabo las pruebas de integración, tendremos que cargar una configuración válida que vayamos a ocupar. De esta forma ya estamos testando el módulo de configuración.

Primeramente, deberemos configurar correctamente los parámetros de conexión, es decir, los puertos para la transmisión de los diferentes datos y la dirección IP del vehículo.

La configuración que utilizaremos para la fase de pruebas es la que aparece en la figura 21. Una vez elegidos dichos valores, guardaremos la configuración.

7.2.1. Software para monitorización

Para realizar las pruebas de este módulo no necesitaremos el vehículo, sino la Raspberry junto con la controladora de vuelo y la cámara del vehículo.

Arrancamos el computador del vehículo. Mediante SSH nos conectamos a él y ejecutamos el módulo Python para monitorizar la orientación. Verificamos que tanto el computador como la controladora de vuelo y la cámara permanecen encendidas.

Grabador y monitor de vídeo.

Comencemos con la visualización de vídeo. Además de la visualización en tiempo real, vamos a testear que el programa puede grabar en disco dichos vídeos. Para ello vamos a configurarlo desde el menú del programa y habilitaremos esa opción.

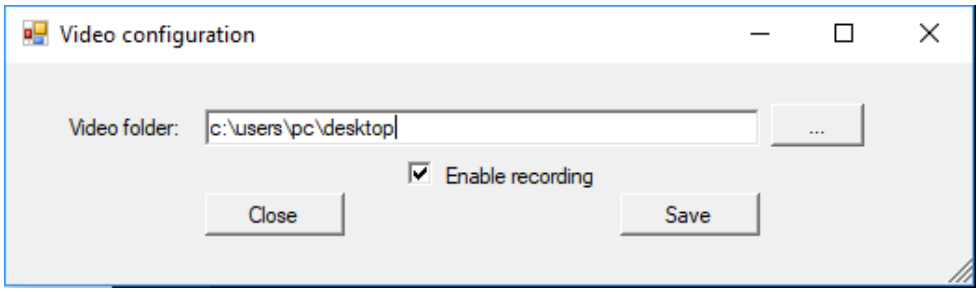

Figura 22. Configuración para grabación de vídeo.

Como vemos en la figura 22 (ventana de configuración para la grabación de vídeo), seleccionamos el directorio donde se generarán los clips, y además habilitaremos la opción de grabación. Una vez hecho esto, guardaremos la configuración deseada.

Mediante SSH ejecutamos el módulo encargado de capturar el vídeo y enviarlo. Desde la Estación nos conectamos y vemos el vídeo. Se observa un retardo casi inapreciable.

Por otra parte, a la hora de verificar la correcta visualización de los clips de vídeo que se han guardado en disco, vemos que se han grabado con un factor de FPS [14] demasiado bajo (~4 fps), lo cual perjudica gravemente la experiencia del usuario a la hora de visualizar dichos clips.

Tras un periodo de depuración de nuestro programa descubrimos el fallo. El problema reside en que el hilo que escribe el vídeo itera hasta ocho veces, mientras que, en ese tiempo, el hilo que está recibiendo las imágenes del vehículo, solo itera una. De esta forma, el hilo escritor escribe hasta ocho veces seguidas el mismo frame repetido al archivo de vídeo.

La solución que proponemos es realizar la escritura del fichero de vídeo en el mismo hilo que recibe las imágenes del vehículo. De esta forma, cada frame solamente podrá ser escrito una única vez.

Monitor wifi.

Por último, ejecutamos el módulo para monitorear la conexión con la red. Desde la Estación nos conectamos y observamos el gráfico de calidad de la señal.

Alejamos el vehículo del punto de acceso a la red y ello produce bajadas de calidad. Igualmente, si lo volvemos a acercar, se observan picos y mejora de la calidad de la señal.

[14] Cuadros por segundo (Frames Per Second) es un indicador de la cantidad de fotogramas que aparecen por segundo en un clip de vídeo. 24 fps es el valor mínimo para “engañar” al ojo humano y hacerle creer que está viendo una película y no una secuencia de imágenes.

7.2.2. Software para pilotaje

En esta fase las pruebas son más críticas, pues vamos a poner en marcha nuestro vehículo. Cualquier fallo en la programación podría provocar que el dron sufriera daños y con ello provocar variaciones en la planificación prevista.

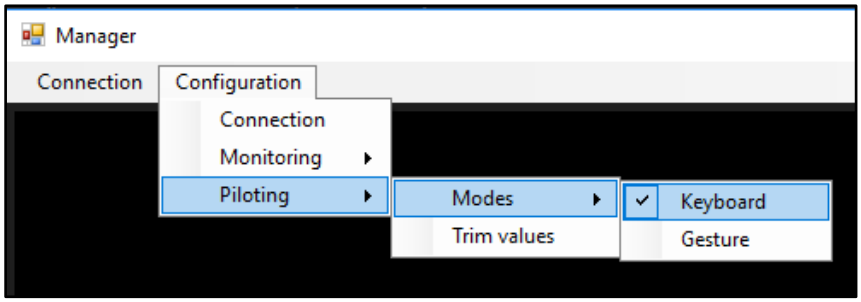

Figura 23. Cambio de modo de pilotaje.

Comenzaremos primeramente a testear el pilotaje mediante teclado. Para ello, como se ve en la figura 23, en la aplicación vamos a seleccionar dicho modo de pilotaje.

Una vez configurado el modo y los valores para la conexión con el computador a bordo, (Configuration > Connection), vamos a conectar con el vehículo (Connection > Connect).

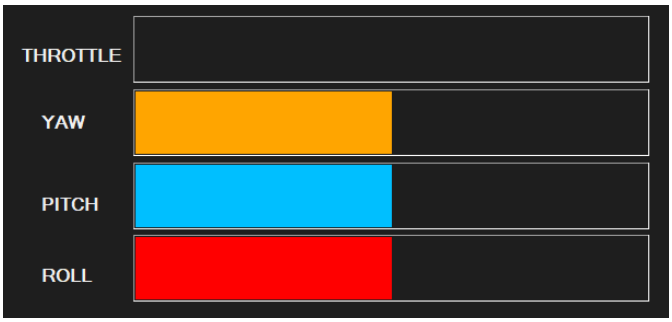

Figura 24. Valores de navegación en reposo tras iniciar la conexión.

Sí todo va bien deberíamos estar visualizando los tres módulos de monitorización correctamente y los valores de navegación deberían encontrarse en posición de reposo (figura 24). Efectivamente por ahora todo lo anterior se cumple.

Tras comenzar a acelerar los motores incrementando el valor de Throttle, manteniendo pulsada la tecla w, vemos que los motores se aceleran, pero el vehículo no se eleva. Ni siquiera hace mención de despegar. Volvemos a comprobar que los motores efectivamente giran en sentido adecuado y las hélices están colocadas correctamente.

Tras haber pesado el vehículo (~600 gr.), deducimos que las hélices no están proporcionando suficiente empuje (hélices bipala con 30º de inclinación en la pala). La solución pasa por cambiar las hélices a unas con mayor inclinación, o incluso con un mayor número de palas.

Una vez hecho el cambio de hélices, volvemos a probar el software. Esta vez el vehículo comienza a despegar rápidamente. El vuelo es estable. De repente una ráfaga de fuerte

viento hace que perdamos el control del vehículo y se estrelle en un árbol. Rápidamente nos damos cuenta de que no hemos habilitado ningún sistema de seguridad durante el vuelo.

Los motores han seguido girando mientras se estaba estrellando debido a que no habíamos habilitado el sistema de seguridad. Esto ha causado que el eje de uno de los motores se doble y quede inutilizable. Así mismo, la Raspberry Pi ha sufrido daños en el circuito tras el impacto y también necesita ser sustituida.

Con las nuevas hélices que hemos integrado (tripala con 45º de inclinación), vamos a proporcionar un empuje más que suficiente. El empuje va a ser tal, que va a provocar que nuestro vehículo sea muy rápido y agresivo cuando aceleremos más de lo debido.

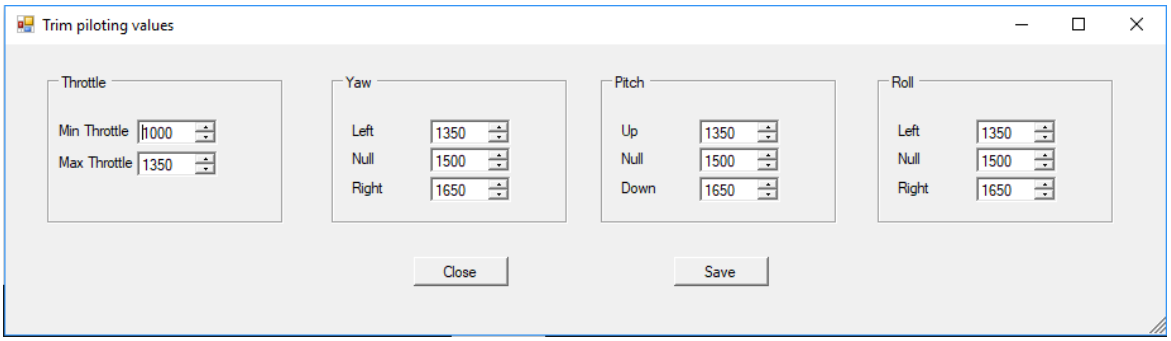

Figura 25. Acotando valor máximo de Throttle.

Por ello vamos a tener que reconfigurar los valores de vuelo y acotaremos el valor máximo de Throttle. De esta manera no alcanzaremos grandes velocidades y nuestro vehículo será más maniobrable y estable.

Tras haber integrado el sistema de seguridad, sustituido los componentes averiados y acotado el valor máximo de Throttle (figura 25), volvemos a hacer la prueba de vuelo y observamos como todo funciona correctamente. Igualmente, con el software de pilotaje mediante gestos obtenemos los mismos resultados.

8. Seguimiento y control

En este apartado veremos cómo se ajusta lo planificado a los hechos que realmente se han producido.

8.1. Duración real de las tareas

TAREA	COMIENZO	FIN	HORAS TOTALES
1. Definición del sistema	**5 de febrero**	**10 de febrero**	**9,5 horas**
1.1. Requisitos previos	5 de febrero	5 de febrero	0,5 horas

1.2. Alcance	5 de febrero	7 de febrero	8 horas
1.3. Dependencias	8 de febrero	8 de febrero	0,5 horas
1.3. Exclusiones	9 de febrero	10 de febrero	0,5 horas
2. Planificación	**12 de febrero**	**21 de febrero**	**23,5 horas**
2.1. Planificación temporal	12 de febrero	17 de febrero	10 horas
2.2. Definición de metodologías	19 de febrero	21 de febrero	12 horas
2.3. Plan de contigencia	21 de febrero	21 de febrero	1,5 horas
3. Análisis del sistema	**22 de febrero**	**26 de febrero**	**22,5 horas**
3.1. Análisis de requisitos	22 de febrero	23 de febrero	7,5 horas
3.2. Diagramas CdU	24 de febrero	8 de marzo	5 horas
3.3. Selección de metodologías	23 de febrero	26 de febrero	10 horas
4. Implementación	**1 de marzo**	**9 de mayo**	**178 horas**
4.1. Configuración y ensamblaje de UAV	1 de marzo	4 de abril	35 horas
4.2. Implementación SW monitorización	10 de marzo	31 de marzo	50 horas
4.3. Implementación módulo de configuración	26 de marzo	5 de abril	3 horas
4.3. Implementación SW pilotaje	5 de abril	9 de mayo	90 horas
5. Pruebas	**7 de mayo**	**15 de junio**	**40 horas**
5.1. Pruebas SW monitorización	7 de mayo	15 de mayo	5 horas
5.2. Pruebas SW pilotaje	15 de mayo	31 de mayo	35 horas
6. Seguimiento y control	**5 de febrero**	**21 de junio**	**27 horas**
6.1. Comparativa tiempos planificados y	5 de febrero	21 de junio	5 horas

reales			
6.2. Cronograma real	5 de febrero	21 de junio	10 horas
6.3. Justificación de desviaciones	9 de marzo	21 de junio	6 horas
6.4. Riesgos materializados	25 de mayo	21 de junio	1 horas
6.5. Reuniones	1 de marzo	21 de junio	8,5 horas
7.Memoria y anexos	**5 de febrero**	**21 de junio**	**32,5 horas**

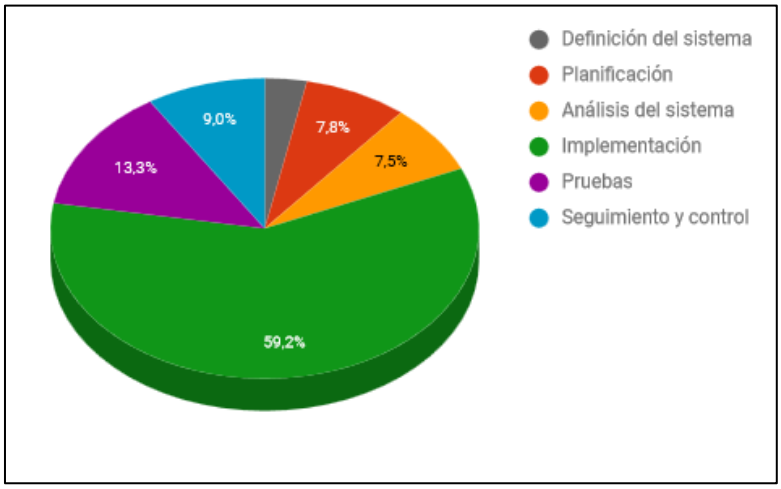

Figura 26. Proporción de dedicación real

8.2. Cronograma real

Tareas		Febrero																
		5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
1.1	Requisitos previos	■																
1.2.	Alcance	■	■	■														
1.3.	Dependencias				■													
1.4.	Exclusiones					■	■											
2.1.	Planificación temporal								■	■	■	■	■	■				
2.2.	Definición de metodologías															■	■	■
2.3.	Plan de contingencia																	■
7.	Documentación y Anexos	■	■	■	■	■	■		■	■	■	■	■	■		■	■	■
6.3.	Justificación desviaciones																	

Tareas		Febrero							Marzo									
		22	23	24	25	26	27	28	1	2	3	4	5	6	7	8	9	10	11
3.1.	Análisis de requisitos																		
3.2.	Diagramas Casos de Uso																		
3.3.	Selección de metodologías																		
4.1.	Config. y ensamblaje de UAV																		
4.2.	Imp. Sw. Monitorización																		
7.	Documentación y Anexos																		
6.3	Justificación desviaciones																		

Tareas		Marzo																
		12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
4.1.	Config. y ensamblaje de UAV																		
4.2.	Imp. Sw. Monitorización																		
4.3.	Imp. Módulo Configuración																		
7.	Documentación y Anexos																		
6.3	Justificación desviaciones																		

Tareas		Marzo		Abril																
		30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
4.1.	Config. y ensamblaje de UAV																					
4.2.	Imp. Sw. Monitorización																					
4.4.	Imp. Sw. Pilotaje																					
4.3.	Imp. Módulo Configuración																					
7.	Documentación y Anexos																					
6.3.	Justificación desviaciones																					

Tareas		Abril											Mayo									
		20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7	8	9	10	11
4.4.	Imp. Sw. Pilotaje																						
5.1.	Pruebas Sw. Monitor.																						
7.	Documentación y Anexos																						
6.3	Justificación desviaciones																						

Tareas		Mayo																				
		12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
5.1.	Pruebas Sw. Monitor.																				
5.2.	Pruebas Sw. Pilotaje																				
7.	Documentación y Anexos																				
6.3	Justificación desviaciones																				

Tareas		Junio																			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
7.	Documentación y Anexos																					
6.3.	Justificación desviaciones																					

8.3. Justificación de desviaciones

En este apartado explicamos cómo difieren los tiempos reales de los tiempos planificados y el porqué.

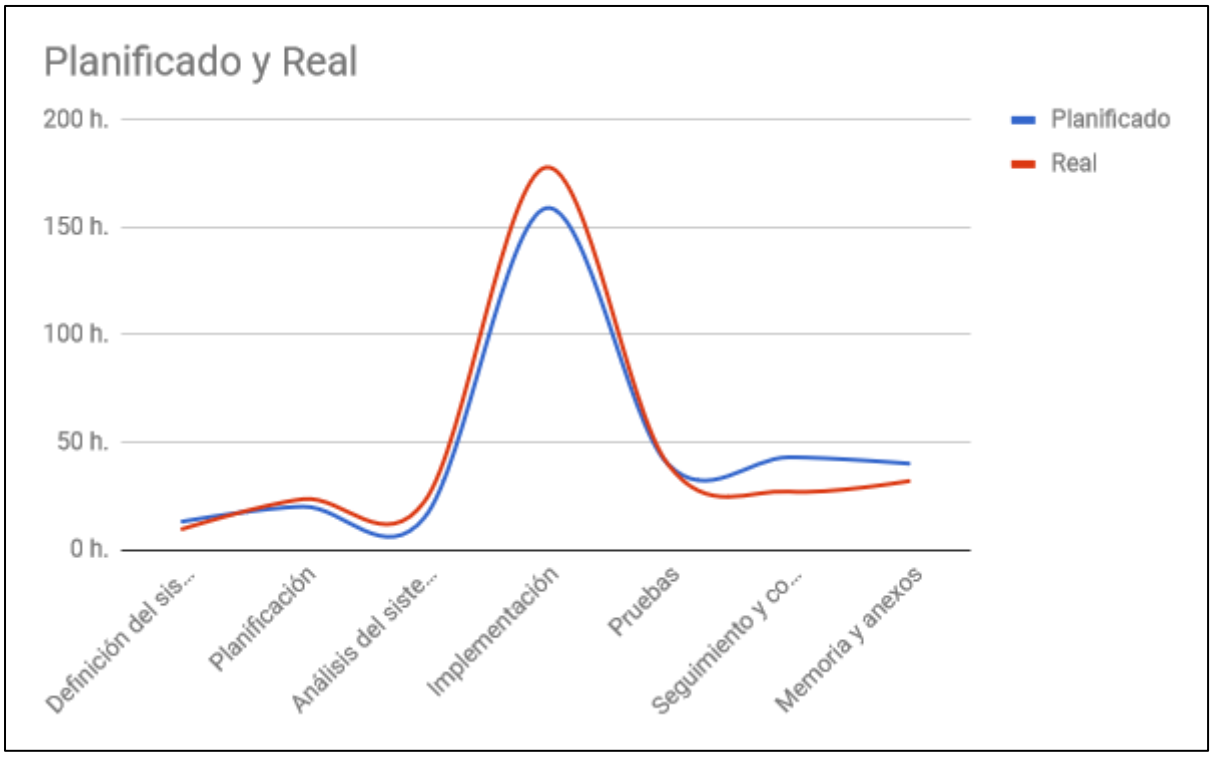

Figura 27. Gráfica comparativa de tiempos.

En la figura 27 tenemos una gráfica comparativa de tiempos fraccionados por fases principales del proyecto. Podemos observar que, por lo general, (hasta la fase de pruebas), la curva de la planificación temporal queda por debajo de la curva que representa la realidad.

Puede ser que, por la falta de experiencia del estudiante en la gestión de proyectos de este tipo, se haya subestimado la distribución temporal para las fases iniciales del proyecto.

En cuanto a la subestimación en la fase de implementación, ha tenido que ver la parte de implementación del software de pilotaje, la cual se ha alargado más de lo que se previó. Hay que tener en cuenta que el estudiante nunca había desarrollado ni pilotado vehículos aéreos. Sin embargo, pese al reto que ha supuesto este proyecto, la desviación temporal no es exagerada.

Finalmente, si nos fijamos en la cola final de la curva (fase de seguimiento y documentación), podemos observar una leve sobreestimación temporal. De nuevo, esto ha sido producido por la inexperiencia del estudiante, principalmente en el ámbito de la redacción de documentos de proyectos como es este.

8.4. Riesgos materializados

Fuente	Riesgo	Causa	Consecuencia	Solución	Coste tiempo
Implementación UAV	Componente hardware dañado	Durante la fase de pruebas del software de pilotaje, un pequeño bug de programación hizo que el UAV se estrellara contra una rama.	Motor número 1 dañado (eje doblado). Raspberry Pi dañada (cortocircuito interno).	Reemplazar motor número 1. Reemplazar Raspberry Pi.	5 horas.
	Componente del vehículo no adecuado	Controladora Naze32 con acelerómetro y giróscopo.	Imposible obtener un valor real de Yaw sin magnetómetro.	Reemplazar por una controladora con magnetómetro.	0,5 horas.
		Hélices bipala con 30º de inclinación.	El UAV no es capaz de despegar por falta de empuje.	Reemplazar por unas hélices tripala con 45º de inclinación.	1 hora.

8.5. Costes de material

A continuación, veremos el coste económico que ha supuesto la adquisición de todo el material para construir el sistema.

Componente	Unidades	Precio/Unidad (€)	Cantidad reutilizada
Chasis	1	14,5	0
Batería LiPo	1	15	0
Motor	5	5,2	0
Hélice	8	2	0

Controladora de vuelo	1	14	1
Placa de distribución	1	3	1
Variadores	4	5	0
Raspberry Pi	2	25	1
Raspberry Pi Camera	1	20	1
Sensor Kinect	1	9,5	1

En total se estima un coste de 188 €. Eliminando el gasto que suponen los componentes reutilizados, en total este proyecto ha supuesto un desembolso de 116,5 €.

9. Conclusiones

9.1. Objetivos alcanzados

Entre los objetivos que nos marcamos en un principio, estaba el de realizar un sistema que nos permitiese pilotar vehículos aéreos no tripulados de diferentes maneras desde el mismo software. Como resultado de este trabajo, además de cumplir con esas capacidades, hemos desarrollado un sistema altamente escalable debido a la modularidad aplicada durante el proceso de desarrollo.

También hemos conseguido monitorizar todos los parámetros del vehículo que propusimos, y hemos dejado el sistema preparado para poder agregar más parámetros, según las prestaciones que ofrezca la controladora de vuelo del vehículo que se utilice en cada caso.

Así mismo, puesto que este proyecto es *código abierto*, dejamos una gran parte del trabajo realizada no solo para nosotros mismos, sino para futuros desarrolladores que quieran bien sea continuar perfeccionando este trabajo o bien reutilizar partes de este proyecto en sus sistemas. Con el fin de facilitar el uso del código fuente a otros desarrolladores, hemos documentado todo el código en inglés.

9.2. Trabajo futuro

Este trabajo fue motivado por el interés que tiene el estudiante en estos vehículos aéreos. Teniendo como finalidad la de desarrollar las bases de un sistema software para gestionar estos vehículos, en un futuro queremos seguir desarrollando y ampliando aún más este sistema. En un futuro cercano al fin de este trabajo, se planea realizar otro modo de pilotaje que pueda ser añadido al software.

En concreto, pretendemos conseguir o desarrollar las bases de un vehículo autónomo que sea capaz de auto comandarse utilizando tecnologías de visión por computador mediante el uso de la cámara que ya hemos instalado en el dron. Por esta razón, para este trabajo ya hemos seleccionado algunas tecnologías como por ejemplo OpenCV, que nos va a facilitar este futuro trabajo.

En cuanto a la parte de la monitorización del vehículo, se pretende ampliar el número de parámetros a monitorear, integrando placas controladoras de vuelo con mejores prestaciones y más sensores.

Por último, debemos destacar la tecnología utilizada para la comunicación. Aunque hemos elegido realizar las comunicaciones vía wifi, este sistema es totalmente compatible con comunicación por radio. Hemos decidido utilizar wifi porque en un futuro queremos integrar estos sistemas (tanto de pilotaje como de monitorización) en dispositivos móviles.

10. Bibliografía

Recurso	Fuente	Disponible en
Python, comunicación UDP	Python Wiki	*https://wiki.python.org/moin/UdpCommunication*
Python, comunicación a bajo nivel con sockets	Python Docs. The Python Standard Library (2018). Low-level networking interface	*https://docs.python.org/3/library/socket.html*
C#, sockets TCP	MSDN	*https://msdn.microsoft.com/es-es/library/system.net.sockets.tcpclient(v=vs.110).aspx*
C#, sockets UDP	MSDN	*https://msdn.microsoft.com/es-es/library/system.net.sockets.udpclient(v=vs.110).aspx*
OpenGL, transformaciones matriciales.	OpenGL Guide (2018), Transformations.	https://open.gl/transformations
OpenGL, conocimientos generales,	OpenGL Superbible (2015). 7th Edition	Libro físico
Blender, conocimientos generales	Blender Docs. 2.79 Reference Manual (2018)	https://docs.blender.org/manual/en/dev/
DesignDoll, conocimientos generales	DesignDoll Terawell Tutorials	https://terawell.net/terawell/?page_id=60
Información sobre Sensores VicoVR	Sitio web de VicoVR	https://vicovr.com/

Información sobre Sensores Orbbec	Sitio web de Orbbec	https://orbbec3d.com/
Comparativa Kinect	Stackoverflow	https://stackoverflow.com/questions/7706448/official-kinect-sdk-vs-open-source-alternatives
OpenTK, información general	Repositorio Github de OpenTK	https://github.com/opentk/opentk
EmguCV, información general	Sitio web de EmguCV	http://www.emgu.com/
Kinect, Información general	MSDN	https://msdn.microsoft.com/en-us/library/hh438998.aspx

11. Anexos

Este trabajo, aparte de este documento, incluye dos anexos. En el primer anexo, se puede apreciar las **actas de las reuniones**, de estudiante y tutores, que han tenido lugar a lo largo de la vida de este proyecto.

El segundo y último anexo, además de ofrecer una breve explicación sobre el funcionamiento interno del sensor de Kinect, analizaremos, seleccionaremos y mapearemos un conjunto de **gestos anatómicos** al conjunto de movimientos del vehículo.